

WEST Search History

DATE: Monday, November 03, 2003

<u>Set Name</u> side by side	<u>Query</u>	<u>Hit Count</u>	<u>Set Name</u> result set
<i>DB=USPT; PLUR=YES; OP=ADJ</i>			
L10	API window32 and spy\$	0	L10
L9	L5 and API	13	L9
L8	L5	30	L8
<i>DB=EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ</i>			
L7	L5	1	L7
<i>DB=PGPB; PLUR=YES; OP=ADJ</i>			
L6	20030001854	1	L6
<i>DB=USPT,PGPB,JPAB,EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ</i>			
L5	L3 and captur\$	40	L5
L4	L3 and inject\$	28	L4
L3	hook\$ and spy	152	L3
<i>DB=USPT,PGPB; PLUR=YES; OP=ADJ</i>			
L2	hooking and spy	13	L2
L1	hooking and spy and inject	3	L1

END OF SEARCH HISTORY

API Spying Techniques for Windows 9x, NT and 2000

Yariv Kaplan

API spying utilities are among the most powerful tools for exploring the inner structure of applications and operating systems. Unfortunately, neither the SDK nor the DDK provide any documentation or examples demonstrating a way for implementing such a utility. This article will attempt to shed some light on this subject by presenting several techniques for hooking API calls issued by Windows applications.

There are several things you'll need to consider before sitting down to write your own API spying utility. First, you'll need to decide whether you want to spy upon a single application or set up a system-wide API interceptor. Each approach can be useful in different situations. For example, assume that you need to write an application that blocks the execution of specific processes according to rules set by an administrator. Obviously, you will need a way to monitor the execution of new processes and terminate the ones, which are marked as restricted. One way to accomplish that would be to establish a system-wide API interceptor that monitors calls made to the `CreateProcess` function (actually, to both the Ansi and Unicode versions of this function). Whenever an application issues a call to one of these functions in order to create a new process, your interceptor will gain control and perform whatever processing is necessary.

Other types of applications might require a simpler API interceptor capable of monitoring just one application at a time. A good example is `BoundsChecker` from NuMega - a tool capable of analyzing API calls in order to detect memory leaks and other bugs lurking inside Windows applications.

Whether you decide to employ a system-wide solution or opt for a simpler one, you'll still have to choose one among several API hooking techniques. The following sections will explore the various ways available for implementing an API interceptor for Windows, focusing on the advantages and disadvantages of each technique.

Proxy DLL

This is by far the easiest technique for hooking API calls under Windows. As an example for a use of this technique, consider an anti-virus application that scans incoming email messages for viruses. An obvious requirement for such an application is the ability to hook Winsock's I/O functions in order to analyze data transfers between email clients and remote mail servers.

This can be easily accomplished by creating a proxy DLL, which contains a stub for each of the functions exported from the Winsock library. If the proxy DLL is named exactly like the original Winsock library (i.e. `wsock32.dll`) and placed in the same directory where the target email application resides, then the interception occurs automatically. When the target application attempts to load the original Winsock library, the proxy DLL is loaded instead. All the calls made to Winsock's functions are routed to the exported stubs in the proxy. After performing the necessary processing, the proxy DLL simply routes the calls to Winsock and returns control back to its caller.

Lest you be afraid that the above method of API interception is too simple, there is a catch. Although simple to implement, this technique has one major drawback - hooking a single function located in a DLL that exports 200 functions would require creating a stub for each of these functions in your proxy DLL. This could be rather tedious and also impossible at times, when some of the functions are not fully documented.

If you wish to see a working example of this technique, consult Matt Pietrek's [WinInet](#) utility, which was

published on the Dec 94 issue of MSJ.

Patch those calls

When thinking about ways for intercepting an API call, there are two locations where you can intervene - either at the source of the call (the application code) or at the destination (the target function). This technique relies on the first option.

For each API function that you wish to intercept, you patch all the locations in the target application where calls to this function are issued. The modification can be done either on disk (to the executable file itself) or in memory (after the executable is already loaded). The tough part is pin pointing the exact locations where patching is necessary. In order to accomplish that, you'll need to implement a disassembler capable of analyzing assembly instructions. Obviously, writing a disassembler is far from being a trivial task, making this API interception technique one of the least popular among the group.

IAT Patching

If you have ever looked into API hooking before, then you probably heard about Import Address Table patching. The numerous advantages of this technique make it the most elegant and common way of hooking API functions under Windows.

The foundation of this technique relies on the fact that 32-bit Windows executable files and DLLs are built upon the Portable Executable (PE) file format. Files based on this specification are composed of several logical chunks known as sections. Each section contains a specific type of content. For example, the .text section holds the compiled code of the application while the .rsrc section serves as a repository for resources such as dialog boxes, bitmaps and toolbars.

Among all of the sections present in a Windows executable file, the .idata section is particularly useful for those who wish to implement an API interceptor. A special table located in this section (known as the Import Address Table) holds file-relative offsets to the names of imported functions referenced by the executable's code. Whenever Windows loads an executable into memory, it patches these offsets with the correct addresses of the imported functions.

Why does Windows go into the trouble of patching the IAT in the first place? Well, as you probably know, Windows executable files and DLLs are often relocated (due to collisions) after they are loaded into memory. This makes it impossible to set in advance the target addresses of calls made to imported functions in the executable code. In order to ensure that these calls successfully reach their destination, it would have been necessary for Windows to locate and patch every single call made to an imported function after an executable image is loaded into memory. Obviously, such a large amount of processing during initializing of new processes and DLLs would have slowed down the system, giving the user the notion as if Windows is a slow and unresponsive operating system ;-).

Fortunately, the designers of Windows were quite resourceful when they addressed this issue. In the current implementation of Windows executables and DLLs, calls made to imported functions are routed through the IAT using an indirect JMP instruction. The fact that imported function calls are "drained" through one location saves Windows the trouble of traversing the executable image in memory, looking for call instructions that are destined for patching.

What all of this got to do with API spying? Well, it seems that Windows IAT redirection mechanism offers a perfect way for intercepting API calls. By overwriting a specific IAT entry with the address of a logging routine, an API interceptor can gain control before the original function gets a chance to be executed by the processor.

Obviously, there are other issues involved in the implementation of this technique, such as the requirement for the logging code to be executed in the memory context of the intercepted application. These issues are discussed in the following resources:

- Matt Pietrek's excellent book: "Windows 95 System Programming Secrets" contains the source code of an API interception utility called APISpy32. This utility was initially published as part of an article written by Matt for the Dec 1995 issue of MSJ. A newer version of APISpy32 is available and can be ordered through Matt's web site at <http://www.wheaty.net>.
- Spying on COM Objects by Dmitri Leman, WDJ, July 1999.

Patch the API

This is my favorite technique for hooking API functions. It has the inherent advantage of being able to trace API calls issued from different parts of an application while requiring a modification only to a single location - the API function itself.

There are several approaches that can be used here. One option is to replace the first byte of target API with a breakpoint instruction (Int 3). Any call issued to that function would generate an exception, which would be reported to your API interceptor in case it serves as a debugger of the target process. Unfortunately, there are several problems with this approach. First, the poor performance of Windows exception handling mechanism would considerably slow down the system. A second problem is related to the implementation of Windows debugging API. As soon as a debugger shuts down, it terminates all the applications that were under its control. Obviously, such a behavior is completely unacceptable in case you're implementing a system-wide interceptor, which must be able to terminate itself before its target applications cease executing.

Another possibility is to patch the target function with one of the control-transfer instructions of the CPU (i.e. either a CALL or a JMP). Once again, there are several problems with this approach. First, it is possible that the patching would overrun the end of the intercepted function. This can occur in cases where the target API is shorter than 5 bytes (CALL and JMP are each 5 bytes long). Another issue is the need to constantly switch between the patched and "unpatched" versions of the intercepted function. This means that once your logging routine receives control from the CPU, it must restore the intercepted function to its previous unhooked state. This is required to allow the API interceptor to route the call to the original function without generating an infinite loop of calls back to the logging routine. Note that during the time the CPU is executing the original function, other calls to that function might be issued from different parts of the system. Since the function is in the unhooked state at that stage, the API interceptor will miss those calls. A more sophisticated API interceptor might utilize a better technique in order to overcome this limitation. Take a look at the following figure to get a better idea how this could be accomplished.

Before Interception:	Following Interception:	
TargetFunction: push ebp mov ebp,esp push ebx push esi push edi ...	TargetFunction: jmp LoggingFunction	LoggingFunction: ... call Stub
	TargetFunction+5: push edi ret
	Stub: push ebp mov ebp,esp push ebx push esi jmp TargetFunction+5 ...	

In this case, the API interceptor places a JMP instruction at the beginning of the target function, but not before saving the first 5 bytes of the function to a pre-allocated buffer in memory (a stub). The exact number of bytes to be copied to the stub may change depending on the instructions present at the head of the function. In cases where 5 bytes do not fall within instruction boundary, it is necessary to copy additional bytes until there is enough space for the JMP instruction to be inserted. Note that relative control-transfer instructions (i.e. JMPs and CALLs) need to be modified during copying to ensure that they transfer control to the right location in memory when executed from the stub. Obviously, performing such an analysis of assembly instructions requires the assistance of a disassembler, which, as I have mentioned before, is not very easy to implement.

If you are not intimidated by the complexity of this technique and wish to use it with your own applications, you might want to have a look at the source code of Detours - an API interception library, which was developed by one of the members in Microsoft's research department.

Breaking address space barriers

By now, you should have a pretty good idea on how to implement an API interceptor capable of redirecting API calls to your own logging code. However, one problem remains unsolved - how do you ensure that the logging code is executed in the right address space? Before we can answer that question, we need to better understand the internal architecture of Windows memory management unit.

As you probably know, each 32-bit Windows application gets a unique address space to toy with. During a task switch, Windows updates its page tables to reflect the new process's linear to physical memory mapping (also known as the process memory context). As a result, the page table entries that correspond to the private memory area of the process are modified to point to different physical addresses while those that correspond to shared memory regions remain untouched.

Under Windows 9x, the 4GB linear address space is divided into several distinct memory regions, each with its own predefined purpose. MS-DOS and the first portion of the 16-bit global heap occupy the lowest 4MB. The next region spans from 4MB to 2GB and it is where Windows 9x loads each process's code, data and DLLs. Since each process physical addresses are unique, Windows 9x ensures that when a specific process is active, the page table entries corresponding to the 4MB-2GB region are mapped to this process's physical memory pages. The idea is for all processes to share the same linear addresses but not the same physical addresses (which is obviously impossible). Think of it as if the 4MB-2GB linear memory region is a window into the physical address space. This window slides each time a process is scheduled for execution to provide a view of the unique physical memory locations occupied by that process.

So what's the real benefit of this mechanism anyway? Well, since all processes use the same linear addresses, Windows can load them into different physical locations without performing any code fix-ups. This means that the memory representation of a process can be (almost) an exact copy of its on-disk image.

Continuing our exploration of the Windows 9x address space reveals that the region spanning from 2GB to 3GB is reserved for the upper portion of the 16-bit global heap. Also in this region are the memory-mapped files and Windows system DLLs (such as USER32, GDI32 and KERNEL32), which are shared among all running processes. This region is extremely useful for API interceptors, since it is visible in all the active address spaces. In fact, APISpy32 loads its spying DLL (APISpy9x.dll) into this area, thus ensuring that its logging code is accessible from any process issuing a call to an intercepted function.

Under Windows NT/2000, the story is a bit different. There is no documented way of loading a DLL into an area shared by all processes, thus the only way to ensure that the logging code is accessible by the

- target process is to inject the spying DLL into its address space. One of the ways to accomplish this is by adding the name of the DLL to the following registry key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_Dlls

This key causes Windows to load your DLL into every address space in the system. Unfortunately, this technique can only be used to inject a DLL into processes that link with user32.dll, meaning that console applications, which do not usually link with this DLL, are not included. Other methods for injecting a DLL into a process's address space are thoroughly described in the article "Load Your 32-bit DLL into Another Process's Address Space Using INJLIB" by Jeffrey Richter, which was published on the May 1994 issue of MSJ. Additional resources are available below:

- Knowledge base articles: [Q197571](#), [Q125680](#), [Q125691](#).

Injecting at the right moment

Knowing how to inject a piece of code into another process's address space is one thing, but timing is also a crucial factor when implementing an API interceptor. Inject at the wrong moment and your interceptor might miss calls issued by the target application. This problem really comes to light when implementing a system-wide interceptor. In that case, the interceptor needs to inject its spying DLL into a process's address space immediately after that process is executed, but right before it issues calls to intercepted functions. The best way to accomplish this is by monitoring calls to the CreateProcess function. When such a call is detected, the interceptor's logging routine passes control to the original CreateProcess function with a modified dwCreationFlags parameter, which contains the CREATE_SUSPENDED value. This ensures that the target process is started, but placed in a suspended state. The interceptor can then inject its spying DLL into the target process's address space and activate it using the ResumeThread API function.

Other ways for detecting the execution of processes under Windows are presented in the following section. Unfortunately, due to their asynchronous nature, they are less suited for detecting the appropriate time for injecting a spying DLL into another process's address space.

Detecting Process Execution

As you saw earlier, it is often necessary for an application to be able to detect the execution of new processes. One possibility, which was previously mentioned, is to hook the CreateProcess function and monitor calls made to that function from different parts of the system. However, implementing a system-wide API interceptor for the sole purpose of hooking a single function often does not justify the effort.

Fortunately, there is a simpler way to accomplish that under Windows 95 (OSR 2 and above), 98 and NT/2000, without requiring the support of a full-fledged system-wide API interceptor. Under Windows 9x, it is possible for a virtual device driver to respond to the CREATE_PROCESS message sent by VWIN32 whenever a new process is executed. Windows NT/2000 offers similar functionality through the use of the undocumented PsSetCreateProcessNotifyRoutine function exported from NTOSKRNL. This function allows a device driver to register a callback function that receives notifications from the operating system whenever a new process is started. If you are well versed in device driver development, you should have no trouble deciphering these interfaces yourself using the following examples as your guide:

- The [Nerditorium](#) column by Jim Finnegan, which was published on the January 1999 issue of MSJ, presented a utility that detects execution of processes under Windows NT/2000.
- The [ProcSpy32](#) utility available on my web site detects execution of processes under Windows 9x using a combination of a device driver and an OCX component.

Winsock Hooking

Many of the programmers that look into API hooking techniques are seeking a way for monitoring network activity performed by Winsock applications. Such functionality is often required by anti-virus utilities, personal firewalls and Internet content blocking applications (e.g. CyberPatrol).

If you require such functionality in your application, you need not write a system-wide API interceptor, but rather use a mechanism, which was introduced along with Winsock 2. This mechanism is thoroughly described in the article Unraveling the Mysteries of Writing a Winsock 2 Layered Service Provider, which appeared in the May 99 issue of MSJ.

An alternative technique for monitoring network activity under Windows relies on the way NDIS drivers are layered on top of each other. By writing an intermediate driver (or alternatively by hooking NDIS interfaces), it is possible to monitor not only TCP/IP communication, but also any other data transferred through the network adapter. Take a look at the following resources for more information about these techniques:

- VToolsD available from NuMega technologies comes bundled with a sample driver (HookTDI), which demonstrates a way for monitoring network activity under Windows 9x.
- IMSAMP is a sample NDIS intermediate driver, available on Microsoft's web site.

DDE and BHO (a.k.a. Browser Helper Object)

In cases where Winsock does not provide sufficient information about the underlying network activity, a programmer can use two additional services, which are tailored specifically for the purpose of monitoring Internet browsers such as Netscape Navigator and Internet Explorer. Through these interfaces it is possible not only to monitor the data transferred through the network, but also to track the browser window, which initiated the network transaction.

Detailed information about these techniques can be found at the following locations:

- Controlling Internet Explorer 4.0 with Browser Helper Objects by Scott Roberts.
- Customizing Microsoft Internet Explorer 5.0 by Dino Esposito.
- Browser Helper Objects: The Browser The Way You Want It by Dino Esposito.
- WebSpy - A visual basic utility that uses DDE to monitor URLs entered into web browsers.
- Netscape DDE documentation
- Knowledge base articles: Q229970, Q167821, Q167826, Q191508, Q160957, Q160976.

GDI Hooking

I often get asked about ways for monitoring graphics operations under Windows. Up until now, there was no documented way to monitor these calls unless you were willing to create a system-wide API interceptor that hooks every GDI function in existence. Obviously, this is far from being an easy solution. Fortunately, the new accessibility API available under Windows 9x introduces a mechanism that allows applications to monitor graphics operations before they reach the video driver. Windows 2000 offers similar functionality through a slightly different interface. Detailed information about these interfaces can be found in the following resources:

- Documentation of the SetDDIHook function available as part of the accessibility API for Windows 9x.
- Knowledge base article Q229664.

File System Monitoring

For information about file system monitoring techniques, check out the following resources:

- Inside the Windows 95 File System by Stan Mitchell.
- Windows NT File System Internals by Rajeew Nagar.
- FileMon - A file system monitoring utility for Windows 9x, NT and 2000.
- E4M - An on-the-fly disk encryption utility that uses a file system filter driver to monitor disk activity.
- "Examining the Windows 95 Layered File System," by Mark Russinovich and Bryce Cogswell, Dr. Dobb's Journal, December 1995.
- "Examining The Windows NT File System," by Mark Russinovich and Bryce Cogswell, Dr. Dobb's Journal, February 1997.

Interrupt Hooking

In the old days of DOS, interrupt hooking was widely used by TSR (Terminate and Stay Resident) utilities and other applications to extend the operating system functionality and monitor its behavior. Under Windows, interrupts still play a major role, and are mainly used as a portal that connects user-mode (a.k.a. ring 3) code to the operating system's kernel. If you wish to hook interrupts under Windows, you should have a look at the following resources:

- Undocumented Windows Nt by Prasad Dabak, Sandeep Phadke and Milind Borate.
- Monitoring NT Debug Services by Jose Flores, Windows Developer Journal, February 2000.
- NTSpy - A utility that monitors NT's system calls by hooking interrupt 2Eh.

COM Hooking

API spying applications are great for monitoring Windows APIs, but their lack of support for COM interfaces makes them useless when trying to monitor OCXs and other OLE components. Fortunately, there is a way for monitoring COM interfaces under Windows. This technique was presented in the article Spying on COM Objects by Dmitri Leman, which was published on the July 1999 issue of WDJ.

16-bit API Interception

Interception of 16-bit code is not common anymore, but some programmers still require such capabilities in their applications. If you are unfortunate enough to be still working on 16-bit code, you might want to have a look at the article: "Hook and Monitor Any 16-bit Windows Function With Our ProcHook DLL" by James Finnegan, which was published on the January 1994 issue of MSJ.

NT System Calls

If you have ever examined ntdll.dll with QuickView, you might have noticed that it exports a set of functions that begin with the Nt prefix. These functions are actually small stubs of code that pass control to the Windows NT kernel (NTOSKRNL) using interrupt 2E. Many of the functions exported from kernel32.dll are nothing more than control transfer routines to the stubs located in ntdll. For example, when a Windows application issues a call to CreateFile located in kernel32.dll, the call is redirected to NtCreateFile, which passes it on to NT's kernel for further processing. The special design of this mechanism allows a device driver to hook these interfaces, thus providing a way for monitoring activities performed by Windows NT/2000 applications. A thorough description of this mechanism is presented in the following resources:

- [Undocumented Windows Nt](#) by Prasad Dabak, Sandeep Phadke and Milind Borate.
- [Windows NT/2000 Native API Reference](#) by Gary Nebbet.
- [Regmon](#) - A utility that monitors access to the registry by using system call hooking techniques.
- [NTSpy](#) - A utility that monitors NT's system calls by hooking interrupt 2Eh.
- [Tracing NT Kernel-Mode Calls](#) by Dmitri Leman, WDJ, April 2000.

Resources

- [Undocumented Windows Nt](#) by Prasad Dabak, Sandeep Phadke and Milind Borate.
- [Windows NT/2000 Native API Reference](#) by Gary Nebbet.
- [Advanced Windows \(3rd Ed\)](#) by Jeffrey Richter.
- [Inside the Windows 95 File System](#) by Stan Mitchell.
- [Windows NT File System Internals](#) by Rajeew Nagar.
- [Windows NT Device Driver Development](#) by Peter Viscarola and Anthony Mason.
- [Developing Windows NT Device Drivers](#) by Edward N. Dekker and Joseph M. Newcomer.
- [Writing Windows Wdm Device Drivers](#) by Chris Cant.
- [Programming the Microsoft Windows Driver Model](#) by Walter Oney and Forrest Foltz.
- [Windows Undocumented File Formats](#) by Pete Davis and Mike Wallace.
- "Windows 95 System Programming Secrets" by Matt Pietrek.



Copyright © 1999, 2000 Yariv Kaplan
yariv@internals.com

All Topics, MFC / C++ >> System >> General

VC6, XP, W2K,
Win9X, MFC

Posted 6 Apr 200

Updated 3 Dec 20

Articles by this au

162,765 views

API hooking revealed

By Ivo Ivanov

The article demonstrates how to build a user mode Win32 API spying system

Search: Articles

[FAQ](#) [What's New](#) [Lounge](#) [Contribute](#) [Message Bo](#)

Toolbox

Broken links?

VS.NET 2003 for \$899

MSDN Univ. from \$1950

Print version

Send to a friend

Download source files - 69 Kb
 Download demo project - 139 Kb

157 users have rated this article. Result:

Popularity: 10.73. Rating: 4.

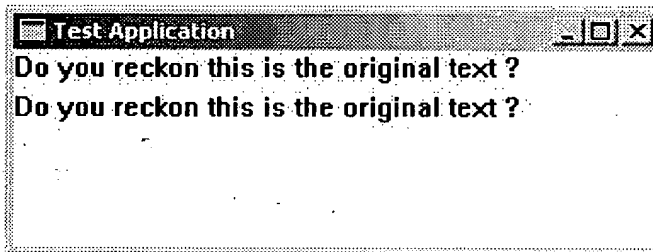
Sign in / Sign up

Email

Password

☒ Remember me

Lost your Password?



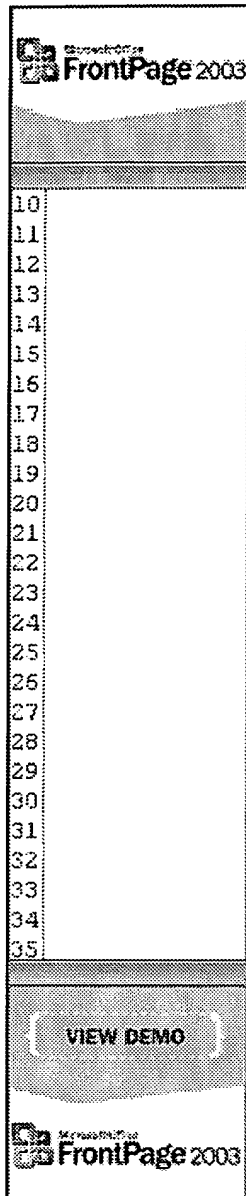
Introduction

Intercepting Win32 API calls has always been a challenging subject among most of the Windows developers and I have to admit, it's been one of my favorite topics. The term represents a fundamental technique of getting control over a particular piece of code execution. It provides a straightforward mechanism that can easily alter the operating system's behavior as well as 3rd party products, without having their source code available.

Many modern systems draw the attention to their ability to utilize existing Windows applications by employing spying techniques. A key motivation for hooking, is not only to contribute to advanced functionalities, but also to inject user-supplied code for debug purposes.

Unlike some relatively "old" operating systems like DOS and Windows 3.xx, the present Windows OS as NT/2K and 9x provide sophisticated mechanisms to separate address space of each process. This architecture offers a real memory protection, thus no application can corrupt the address space of another process or in the worse case even to crash the operating system itself. This fact makes a lot harder the development of system-aware hooks.

My motivation for writing this article was the need for a really simple hooking framework that will offer an easy to use interface and ability to capture different APIs. It intends to reveal some of the tricks that can help you to write your own spying system. It suggests a solution how to build a set for hooking Win32 API functions on NT/2K as well as 98/M (named in the article 9x) family Windows. For the sake of simplicity I decided not to support UNICODE. However, with some minor modifications of the code you could accomplish this task.



Spying of applications provides many advantages:

1. **API function's monitoring**

The ability to control API function calls is extremely helpful and enables developers track down specific "invisible" actions that occur during the API call. It contributes comprehensive validation of parameters as well as reports problems that are usually overlooked behind the scene. For instance, sometimes, it might be very helpful to monitor memory-related API functions for catching resource leaks.

2. **Debugging and reverse engineering**

Besides the standard methods for debugging, API hooking has a deserved reputation being one of the most popular debugging mechanisms. Many developers employ hooking technique in order to identify different component implementations and relationships. API interception is a very powerful way of getting information about executable.

3. **Peering inside operating system**

Often developers are keen to know operating system in depth and are inspired by being a "debugger". Hooking is also quite a useful technique for decoding undocumented or poorly documented APIs.

4. **Extending originally offered functionalities** by embedding custom modules

External Windows applications re-routing the normal code execution by injection can provide an easy way to change and extend existing module functionalities. For example, many 3rd party products sometimes don't meet specific security requirements and have to be adjusted to your specific needs. Spying of applications allows developers to add sophisticated pre- and post-processing around the original API functions. This ability is extremely useful for altering the behavior of the already compiled code.

Functional requirements of a hooking system

There are a few important decisions that have to be made, before you start implementing a kind of API hooking system. First of all, you should determine whether to hook a single application or to install a system-aware engine. For instance, if you would like to monitor one application, you don't need to install a system-wide hook, but if your job is to track all calls to `TerminateProcess()` or `WriteProcessMemory()`, the only way to do so is to install a system-aware hook. What approach you will choose depends on the particular situation and addresses specific problems.

General design of an API spying framework

Usually a Hook system is composed of at least two parts - a Hook Server and a Driver. The Hook Server is responsible for injecting the Driver into targeted processes at the appropriate moment. It also administers the driver and optionally can receive information from the driver about its activities, whereas the Driver module performs the actual interception. This design is rough and beyond doubt doesn't cover all possible implementations. It outlines the boundaries of a hook framework.

Once you have the requirement specification of a hook framework, there are a few design issues you should take into account:

- What applications do you need to hook
- How to inject the DLL into targeted processes or which implanting technique to use
- Which interception mechanism to use

I hope the next few sections will provide answers to those issues.

Injecting techniques

1. Registry

In order to inject a DLL into processes that link with USER32.DLL, you simply ca the DLL name to the value of the following registry key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs

Its value contains a single DLL name or group of DLLs separated either by comm spaces. According to MSDN documentation [7], all DLLs specified by the value o key are loaded by each Windows-based application running within the current lo session. It is interesting that the actual loading of these DLLs occurs as a part o USER32's initialization. USER32 reads the value of mentioned registry key and LoadLibrary() for these DLLs in its DllMain code. However this trick applies o applications that use USER32.DLL. Another restriction is that this built-in mecha supported only by NT and 2K operating systems. Although it is a harmless way a DLL into a Windows processes there are few shortcomings:

- o In order to activate/deactivate the injection process you have to reboot W
- o The DLL you want to inject will be mapped only into these processes that USER32.DLL, thus you cannot expect to get your hook injected into conso applications, since they usually don't import functions from USER32.DLL.
- o On the other hand you don't have any control over the injection process. that it is implanted into every single GUI application, regardless you want It is a redundant overhead especially if you intend to hook few application For more details see [2] "Injecting a DLL Using the Registry"

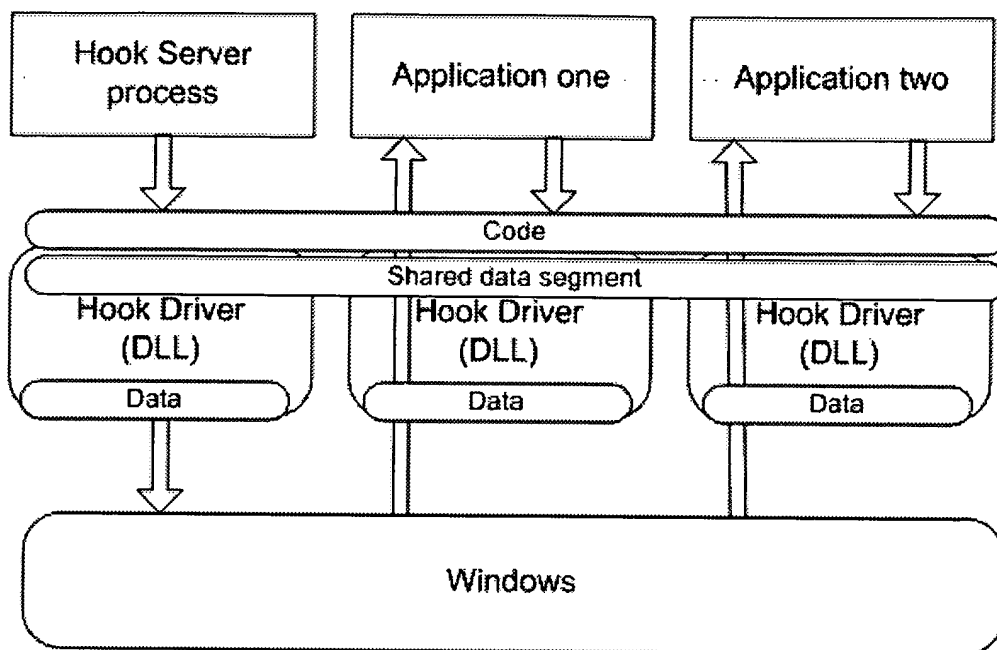
2. System-wide Windows Hooks

Certainly a very popular technique for injecting DLL into a targeted process reli provided by Windows Hooks. As pointed out in MSDN a hook is a trap in the sys message-handling mechanism. An application can install a custom filter function monitor the message traffic in the system and process certain types of message they reach the target window procedure.

A hook is normally implemented in a DLL in order to meet the basic requiremen system-wide hooks. The basic concept of that sort of hooks is that the hook cal procedure is executed in the address spaces of each hooked up process in the s To install a hook you call SetWindowsHookEx() with the appropriate parameters the application installs a system-wide hook, the operating system maps the DLL address space in each of its client processes. Therefore global variables within t will be "per-process" and cannot be shared among the processes that have load hook DLL. All variables that contain shared data must be placed in a shared dat The diagram bellow shows an example of a hook registered by Hook Server and into the address spaces named "Application one" and "Application two".

Figure 1

Windows Hooks



A system-wide hook is registered just ones when `SetWindowsHookEx()` is executed. If an error occurs a handle to the hook is returned. The returned value is required at the custom hook function when a call to `CallNextHookEx()` has to be made. After a successful call to `SetWindowsHookEx()`, the operating system injects the DLL automatically (but not necessary immediately) into all processes that meet the requirements for this particular hook filter. Let's have a closer look at the following dummy `WH_GETMESSAGE` filter function:

```
//-----
// GetMsgProc
//
// Filter function for the WH_GETMESSAGE - it's just a dummy function
//-----
LRESULT CALLBACK GetMsgProc(
    int code,          // hook code
    WPARAM wParam,     // removal option
    LPARAM lParam      // message
)
{
    // We must pass the all messages on to CallNextHookEx.
    return ::CallNextHookEx(sg_hGetMsgHook, code, wParam, lParam);
}
```

A system-wide hook is loaded by multiple processes that don't share the same space.

For instance hook handle `sg_hGetMsgHook`, that is obtained by `SetWindowsHook` is used as parameter in `CallNextHookEx()` must be used virtually in all address. It means that its value must be shared among hooked processes as well as the Server application. In order to make this variable "visible" to all processes we store it in the shared data section.

The following is an example of employing `#pragma data_seg()`. Here I would like to mention that the data within the shared section must be initialized, otherwise the variables will be assigned to the default data segment and `#pragma data_seg()` have no effect.

```
//-----
// Shared by all processes variables
//-----
#pragma data_seg(".HKT")
HHOOK sg_hGetMsgHook      = NULL;
BOOL  sg_bHookInstalled   = FALSE;
// We get this from the application who calls SetWindowsHookEx()'s wrapper
HWND  sg_hwndServer        = NULL;
#pragma data_seg()
```

You should add a `SECTIONS` statement to the DLL's DEF file as well

```
SECTIONS
{
    .HKT    Read Write Shared
}
```

or use

```
#pragma comment(linker, "/section:.HKT, rws")
```

Once a hook DLL is loaded into the address space of the targeted process, there is a way to unload it unless the Hook Server calls `UnhookWindowsHookEx()` or the hook application shuts down. When the Hook Server calls `UnhookWindowsHookEx()` the operating system loops through an internal list with all processes which have been loaded the hook DLL. The operating system decrements the DLL's lock count and when it becomes 0, the DLL is automatically unmapped from the process's address space.

Here are some of the advantages of this approach:

- o This mechanism is supported by NT/2K and 9x Windows family and hopefully will be maintained by future Windows versions as well.
- o Unlike the registry mechanism of injecting DLLs this method allows DLL to be unloaded when Hook Server decides that DLL is no longer needed and makes a call to `UnhookWindowsHookEx()`

Although I consider Windows Hooks as very handy injection technique, it comes with its own disadvantages:

- o Windows Hooks can degrade significantly the entire performance of the system because they increase the amount of processing the system must perform for each message.
- o It requires a lot of efforts to debug system-wide Windows Hooks. However, more than one instance of VC++ running at the same time, it would simplify the debugging process for more complex scenarios.
- o Last but not least, this kind of hooks affects the processing of the whole system under certain circumstances (say a bug) you must reboot your machine to recover it.

3. Injecting DLL by using `CreateRemoteThread()` API function

Well, this is my favorite one. Unfortunately it is supported only by NT and Windows operating systems. It is bizarre, that you are allowed to call (link with) this API 9x as well, but it just returns NULL without doing anything.

Injecting DLLs by remote threads is Jeffrey Richter's idea and is well documented in his article [9] "Load Your 32-bit DLL into Another Process's Address Space Using IN

The basic concept is quite simple, but very elegant. Any process can load a DLL dynamically using LoadLibrary() API. The issue is how do we force an external process to call LoadLibrary() on our behalf, if we don't have any access to the process's thread. Well, there is a function, called CreateRemoteThread() that addresses creating a remote thread. Here comes the trick - have a look at the signature of the thread function, where a pointer is passed as parameter (i.e. LPTHREAD_START_ROUTINE) to the CreateRemoteThread():

```
DWORD WINAPI ThreadProc(LPVOID lpParameter);
```

And here is the prototype of LoadLibrary API

```
HMODULE WINAPI LoadLibrary(LPCTSTR lpFileName);
```

Yes, they do have "identical" pattern. They use the same calling convention, both accept one parameter and the size of the returned value is the same. This may give us a hint that we can use LoadLibrary() as the thread function, which will be executed when the remote thread has been created. Let's have a look at the following sample code:

```
hThread = ::CreateRemoteThread(  
    hProcessForHooking,  
    NULL,  
    0,  
    pfnLoadLibrary,  
    "C:\\HookTool.dll",  
    0,  
    NULL);
```

By using GetProcAddress() API we get the address of the LoadLibrary() API. A dodgy thing here is that Kernel32.DLL is mapped always to the same address in each process, thus the address of LoadLibrary() function has the same value in the address space of any running process. This ensures that we pass a valid pointer (i.e. pfnLoadLibrary) as parameter of CreateRemoteThread().

As parameter of the thread function we use the full path name of the DLL, cast to LPVOID. When the remote thread is resumed, it passes the name of the DLL to the ThreadFunction (i.e. LoadLibrary). That's the whole trick with regard to using threads for injection purposes.

There is an important thing we should consider, if implanting through the CreateRemoteThread() API. Every time before the injector application operates on the virtual memory of the targeted process and makes a call to CreateRemoteThread(), it first opens the process using OpenProcess() API and passes PROCESS_ALL_ACCESS

as parameter. This flag is used when we want to get maximum access rights to process. In this scenario `OpenProcess()` will return `NULL` for some of the processes with low ID number. This error (although we use a valid process ID) is caused by no security context that has enough permissions. If you think for a moment you will realize that it makes perfect sense. All those restricted processes are part of the operating system and a normal application shouldn't be allowed to operate on them. What would happen if some application has a bug and accidentally attempts to operate on an operating system's process? To prevent the operating system from that kind of eventual crashes, it is required that a given application must have sufficient privileges to execute APIs that might alter operating system behavior. To get access to the system resources (e.g. `smss.exe`, `winlogon.exe`, `services.exe`, etc) through `OpenProcess` invocation, you must be granted the debug privilege. This ability is extremely powerful and offers a way to access the system resources, that are normally restricted. Acquiring the process privileges is a trivial task and can be described with the following operations:

- o Open the process token with permissions needed to adjust privileges
- o Given a privilege's name "`SeDebugPrivilege`", we should locate its local mapping. The privileges are specified by name and can be found in Platform file `winnt.h`
- o Adjust the token in order to enable the "`SeDebugPrivilege`" privilege by `AdjustTokenPrivileges()` API
- o Close obtained by `OpenProcessToken()` process token handle

For more details about changing privileges see [10] "Using privilege".

4. **Implanting through BHO add-ins**

Sometimes you will need to inject a custom code inside Internet Explorer only. Fortunately Microsoft provides an easy and well documented way for this purpose: Browser Helper Objects. A BHO is implemented as COM DLL and once it is properly registered, each time when IE is launched it loads all COM components that have implemented `IObjectWithSite` interface.

5. **MS Office add-ins**

Similarly, to the BHOs, if you need to implant in MS Office applications code of your own, you can take the advantage of provided standard mechanism by implementing add-ins. There are many available samples that show how to implement this kind of add-ins.

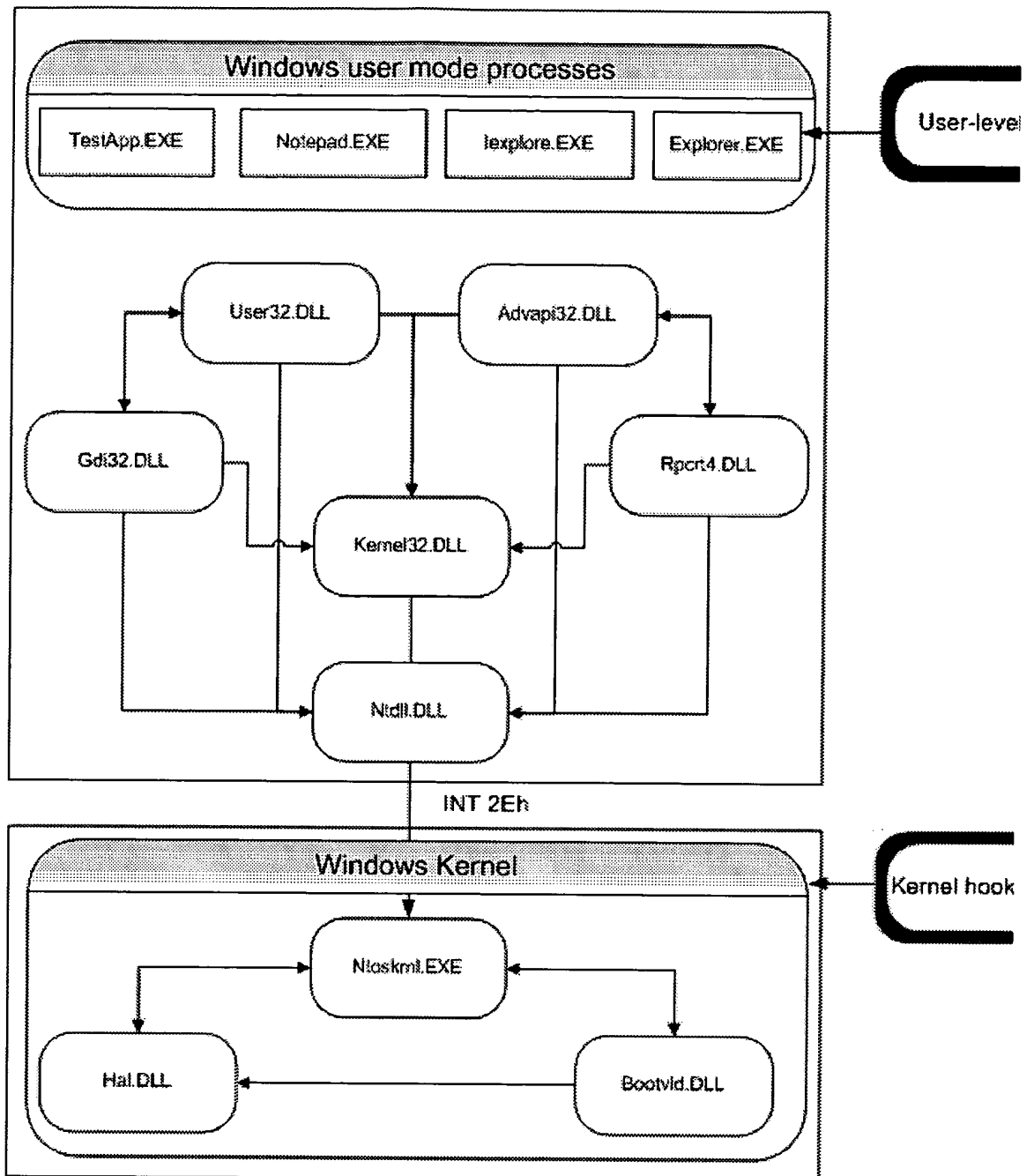
Interception mechanisms

Injecting a DLL into the address space of an external process is a key element of a spy system. It provides an excellent opportunity to have a control over process's threads. However it is not sufficient to have the DLL injected if you want to intercept API functions within the process.

This part of the article intends to make a brief review of several available real-world hooks. It focuses on the basic outline for each one of them, exposing their advantages and disadvantages.

In terms of the level where the hook is applied, there are two mechanisms for API spying: Kernel level and User level spying. To get better understanding of these two levels you should be aware of the relationship between the Win32 subsystem API and the Native API. Figure 2 demonstrates where the different hooks are set and illustrates the module relationships and their dependencies on Windows 2K:

Figure 2



The major implementation difference between them is that interceptor engine for kernel hooking is wrapped up as a kernel-mode driver, whereas user-level hooking usually uses user-mode DLL.

1. NT Kernel level hooking

There are several methods for achieving hooking of NT system services in kernel. The most popular interception mechanism was originally demonstrated by Mark Russinovich and Bryce Cogswell in their article [3] "Windows NT System-Call Hooking". Their basic idea is to inject an interception mechanism for monitoring NT system calls just below the user mode. This technique is very powerful and provides an extremely flexible method for hooking the point that all user-mode threads pass through before they are serviced by the OS kernel.

You can find an excellent design and implementation in "Undocumented Window Secrets" as well. In his great book Sven Schreiber explains how to build a kernel hooking framework from scratch [5].

Another comprehensive analysis and brilliant implementation has been provided Prasad Dabak in his book "Undocumented Windows NT" [17].

However, all these hooking strategies, remain out of the scope of this article.

2. Win32 User level hooking

a. Windows subclassing.

This method is suitable for situations where the application's behavior might be changed by a new implementation of the window procedure. To accomplish this you simply call `SetWindowLongPtr()` with `GWLP_WNDPROC` parameter and a pointer to your own window procedure. Once you have the new subclass set up, every time when Windows dispatches a message to a specified window it looks for the address of the window's procedure associated with the particular window and calls your procedure instead of the original one.

The drawback of this mechanism is that subclassing is available only with the boundaries of a specific process. In other words an application should not create a window class created by another process.

Usually this approach is applicable when you hook an application through (i.e. DLL / In-Proc COM component) and you can obtain the handle to the window whose procedure you would like to replace.

For example, some time ago I wrote a simple add-in for IE (Browser Help) that replaces the original pop-up menu provided by IE using subclassing.

b. Proxy DLL (Trojan DLL)

An easy way for hacking API is just to replace a DLL with one that has the same name and exports all the symbols of the original one. This technique can be effortlessly implemented using function forwarders. A function forwarder is an entry in the DLL's export section that delegates a function call to another DLL's function.

You can accomplish this task by simply using `#pragma comment`:

```
#pragma comment(linker, "/export:DoSomething=DllImpl.ActuallyDoSomething")
```

However, if you decide to employ this method, you should take the responsibility of providing compatibilities with newer versions of the original library. For more details see [13a] section "Export forwarding" and [2] "Function Forwarder".

c. Code overwriting

There are several methods that are based on code overwriting. One of them changes the address of the function used by CALL instruction. This method is difficult, and error prone. The basic idea beneath is to track down all CALL instructions in the memory and replace the addresses of the original function with the user supplied one.

Another method of code overwriting requires a more complicated implementation. Briefly, the concept of this approach is to locate the address of the original function and to change the first few bytes of this function with a JMP instruction that redirects the call to the custom supplied API function. This method is extremely tricky and involves a sequence of restoring and hooking operations for each individual call. It's important to point out that if the function is in unhooked state and another call is made during that stage, the system won't be able to catch that second call.

The major problem is that it contradicts with the rules of a multithreaded environment.

However, there is a smart solution that solves some of the issues and provides a sophisticated way for achieving most of the goals of an API interceptor. If you are interested you might peek at [12] Detours implementation.

d. **Spying by a debugger**

An alternative to hooking API functions is to place a debugging breakpoint on the target function. However there are several drawbacks for this method. The issue with this approach is that debugging exceptions suspend all application threads. It requires also a debugger process that will handle these exceptions. Another problem is caused by the fact that when the debugger terminates the debugger is automatically shut down by Windows.

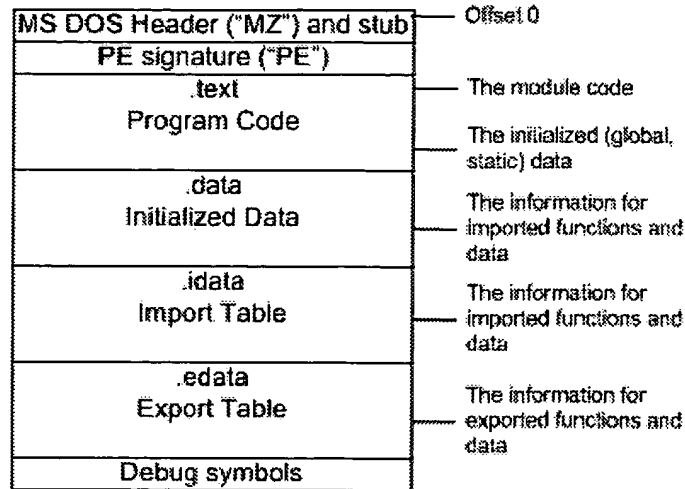
e. **Spying by altering of the Import Address Table**

This technique was originally published by Matt Pietrek and then elaborated by Jeffrey Richter ([2] "API Hooking by Manipulating a Module's Import Section") and John Robbins ([4] "Hooking Imported Functions"). It is very robust, simple and quite easy to implement. It also meets most of the requirements of a hooking framework that targets Windows NT/2K and 9x operating systems. The core of this technique relies on the elegant structure of the Portable Executable (PE) file format. To understand how this method works, you should begin with some of the basics behind PE file format, which is an extension of the COFF (Common Object File Format) format. Matt Pietrek reveals the PE format in details in his wonderful articles - [6] "Peering Inside the PE.", and [13a/b] "An In-Depth Look into the Win32 PE file format". I will give you a brief overview of the PE specification, just enough to get the idea of hooking by manipulation of the Import Address Table.

In general an PE binary file is organized, so that it has all code and data sections. The layout that conforms to the virtual memory representation of an executable file format is composed of several logical sections. Each of them maintains a particular type of data and addresses particular needs of the OS loader.

The section `.idata`, I would like to focus your attention on, contains information about the Import Address Table. This part of the PE structure is particularly crucial for building a spy program based on altering the IAT. Each executable that conforms with the PE format has a layout roughly described in the figure below.

Figure 3



The program loader is responsible for loading an application along with all DLLs into the memory. Since the address where each DLL is loaded into, known in advance, the loader is not able to determine the actual address of the imported function. The loader must perform some extra work to ensure that the program will call successfully each imported function. But going through the executable image in the memory and fixing up the addresses of all imported functions one by one would take an unreasonable amount of processing time and cause huge performance degradation. So, how does the loader resolve this challenge? The key point is that each call to an imported function must be dispatched to the same address, where the function code resides in the DLL. Each call to an imported function is in fact an indirect call, routed through an indirect JMP instruction. The benefit of this design is that the loader does not have to search through the whole image of the file. The solution appears quite simple - it just fixes-up the addresses of all imports inside the IAT. Figure 4 shows an example of a snapshot of the PE File structure of a simple Win32 Application, taken with the help of the [8] PEView utility. As you can see, the TestApp import table contains two imported functions from GDI32.DLL - TextOutA() and GetStockObject().

Figure 4

PEView - C:\Dev\Soft\Magazines\Wd\YAPI\hooking\Code\Output\TestApp.exe

File Edit View Go Help

TestApp.exe

	pFile	Data	Description	Value
IMAGE_DOS_HEADER	00028210	0002A408	HintName RVA	020C TestOutA
MS-DOS Stub Program	00028214	0002A406	HintName RVA	0167 GetStackOb
IMAGE_NT_HEADERS	00028218	00000000	End of Imports	0002 d8
IMAGE_SECTION_HEADER .text	00028244	0002A612	HintName RVA	020B UnhandledE
IMAGE_SECTION_HEADER .idata	00028248	0002A643	HintName RVA	00C2 FreeEmmrom
IMAGE_SECTION_HEADER .data	0002824C	0002A662	HintName RVA	0301 WideCharTo
IMAGE_SECTION_HEADER .idata	00028250	0002A656	HintName RVA	00E9 FlushFileBu
IMAGE_SECTION_HEADER .reloc	00028254	0002A645	HintName RVA	02A8 SetStdHand
SECTION .text	00028258	0002A66A	HintName RVA	001E CloseHandl
SECTION .idata	0002825C	0002A622	HintName RVA	016C GetStringTy
SECTION .data	00028260	0002A610	HintName RVA	0169 GetStringTy
SECTION .idata	00028264	0002A634	HintName RVA	0295 SetFilePoint
IMPORT Directory Table	00028268	0002A770	HintName RVA	010C LCMMapStrin
IMPORT Lookup Table	0002826C	0002A7DA	HintName RVA	0202 MultiByteTo
IMPORT Address Table	00028270	0002A800	HintName RVA	010D LCMMapStrin
IMPORT HintName Table & DLL N	00028274	0002A75C	HintName RVA	02EE VirtualAlloc
SECTION .reloc	00028278	0002A7B0	HintName RVA	01B4 HeapAlloc
IMAGE_DEBUG_TYPE_CODEVIEW	0002827C	0002A7CC	HintName RVA	01B0 HeapReAlloc
	00028280	0002A4EE	HintName RVA	013A GetModuleH
	00028284	0002A602	HintName RVA	0166 GetStartup
	00028288	0002A514	HintName RVA	00DA GetCommarr
	0002828C	0002A626	HintName RVA	018E GetVersion
	00028290	0002A534	HintName RVA	008C ExitProcess
	00028294	0002A542	HintName RVA	0055 DebugStea
	00028298	0002A650	HintName RVA	0168 SetStdHand

Viewing IMPORT Address Table

Actually the hooking process of an imported function is not that complex at first sight. In a nutshell an interception system that uses IAT patching discover the location that holds the address of imported function and repl with the address of a user supplied function by overwriting it. An import requirement is that the newly provided function must have exactly the sa signature as the original one. Here are the logical steps of a replacing cyc

- Locate the import section from the IAT of each loaded by the proces module as well as the process itself
- Find the IMAGE_IMPORT_DESCRIPTOR chunk of the DLL that exports function. Practically speaking, usually we search this entry by the n the DLL
- Locate the IMAGE_THUNK_DATA which holds the original address of th imported function
- Replace the function address with the user supplied one

By changing the address of the imported function inside the IAT, we ensu calls to the hooked function will be re-routed to the function interceptor.

Replacing the pointer inside the IAT is that .idata section doesn't necess to be a writable section. This requires that we must ensure that .idata s can be modified. This task can be accomplished by using VirtualProtect

Another issue that deserves attention is related to the GetProcAddress() behavior on Windows 9x system. When an application calls this API outsid debugger it returns a pointer to the function. However if you call this func within from the debugger it actually returns different address than it wou when the call is made outside the debugger. It is caused by the fact that inside the debugger each call to GetProcAddress() returns a wrapper to pointer. Returned by GetProcAddress() value points to PUSH instruction

by the actual address. This means that on Windows 9x when we loop through, we must check whether the address of examined function is a PUSH instruction (0x68 on x86 platforms) and accordingly get the proper value of the address function.

Windows 9x doesn't implement copy-on-write, thus operating system attempts to keep away the debuggers from stepping into functions above the 2-GB mark. That is the reason why `GetProcAddress()` returns a debug thunk instead of the actual address. John Robbins discusses this problem in [4] "Hooking Important Functions".

Figuring out when to inject the hook DLL

That section reveals some challenges that are faced by developers when the selected mechanism is not part of the operating system's functionality. For example, performing the injection is not your concern when you use built-in Windows Hooks in order to implement a hook DLL. It is an OS's responsibility to force each of those running processes that meet the requirements for this particular hook, to load the DLL [18]. In fact Windows keeps track of newly launched processes and forces them to load the hook DLL. Managing injection registry is quite similar to Windows Hooks. The biggest advantage of all those "built-in" methods is that they come as part of the OS.

Unlike the discussed above implanting techniques, to inject by `CreateRemoteThread()` requires maintenance of all currently running processes. If the injecting is made not optimal, this can cause the Hook System to miss some of the calls it claims as intercepted. It is that the Hook Server application implements a smart mechanism for receiving notifications each time when a new process starts or shuts down. One of the suggested methods in this case, is to intercept `CreateProcess()` API family functions and monitor all their invocations. Thus when an user supplied function is called, it can call the original `CreateProcess()` with `dwCreationFlags` OR-ed with `CREATE_SUSPENDED` flag. This means that the primary thread of the targeted application will be in suspended state, and the Hook Server will have the opportunity to inject the DLL by hand-coded machine instructions and resume the application using `ResumeThread()` API. For more details you might refer to [2] "Injecting Code with `CreateProcess()`".

The second method of detecting process execution, is based on implementing a simple driver. It offers the greatest flexibility and deserves even more attention. Windows NT provides a special function `PsSetCreateProcessNotifyRoutine()` exported by NTOS. This function allows adding a callback function, that is called whenever a process is created or deleted. For more details see [11] and [15] from the reference section.

Enumerating processes and modules

Sometimes we would prefer to use injecting of the DLL by `CreateRemoteThread()` API especially when the system runs under NT/2K. In this case when the Hook Server is started, it must enumerate all active processes and inject the DLL into their address spaces. Windows NT and Windows 2K provide a built-in implementation (i.e. implemented by `Kernel32.dll`) `EnumProcessModules()` Help Library. On the other hand Windows NT uses for the same purpose PSAPI library. Both need a way to allow the Hook Server to run and then to detect dynamically which process "helper" is available. Thus the system can determine which the supported library is, and accordingly to use the appropriate APIs.

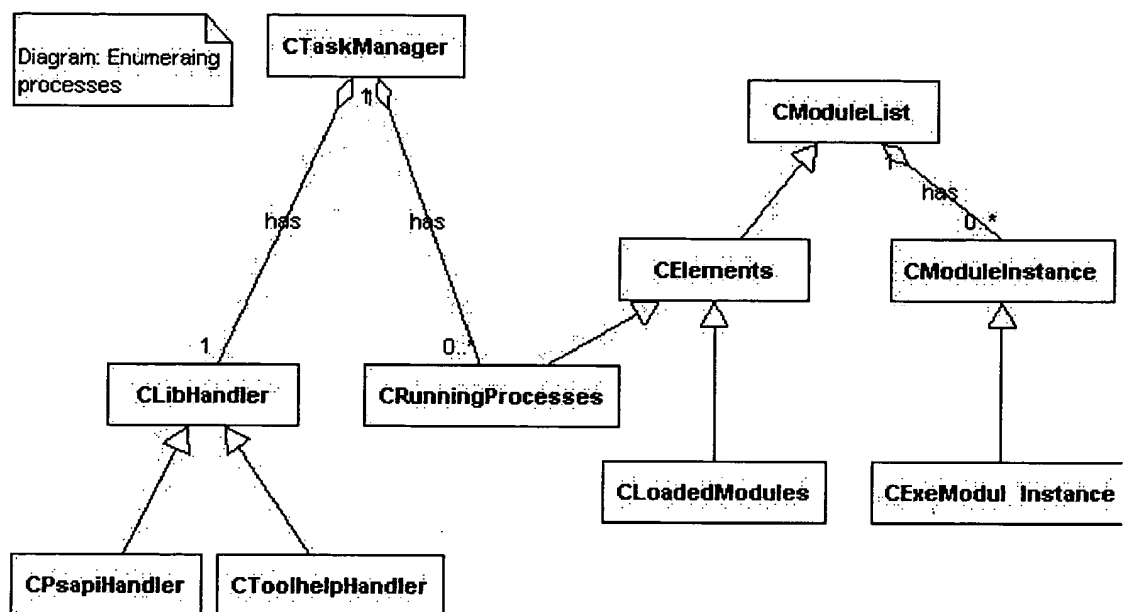
I will present an object-oriented architecture that implements a simple framework for

retrieving processes and modules under NT/2K and 9x [16]. The design of my classes extending the framework according to your specific needs. The implementation itself straightforward.

`CTaskManager` implements the system's processor. It is responsible for creating an in a specific library handler (i.e. `CPSapiHandler` or `CToolhelpHandler`) that is able to e correct process information provider library (i.e. `PSAPI` or `ToolHelp32` respectively). `CTaskManager` is in charge of creating and marinating a container object that keeps a all currently active processes. After instantiating of the `CTaskManager` object the appl calls `Populate()` method. It forces enumerating of all processes and DLL libraries and them into a hierarchy kept by `CTaskManager`'s member `m_pProcesses`.

Following UML diagram shows the class relationships of this subsystem:

Figure 5



It is important to highlight the fact that NT's `Kernel32.dll` doesn't implement any of th `ToolHelp32` functions. Therefore we must link them explicitly, using runtime dynamic we use static linking the code will fail to load on NT, regardless whether or not the ap has attempted to execute any of those functions. For more details see my article "Sin interface for enumerating processes and modules under NT and Win9x/2K."

Requirements of the Hook Tool System

Now that I've made a brief introduction to the various concepts of the hooking proces time to determine the basic requirements and explore the design of a particular hook system. These are some of the issues addressed by the Hook Tool System:

- Provide a user-level hooking system for spying any Win32 API functions import name
- Provide the abilities to inject hook driver into all running processes by Windows

well as `CreateRemoteThread()` API. The framework should offer an ability to se by an INI file

- Employ an interception mechanism based on the altering Import Address Table
- Present an object-oriented reusable and extensible layered architecture
- Offer an efficient and scalable mechanism for hooking API functions
- Meet performance requirements
- Provide a reliable communication mechanism for transferring data between the and the server
- Implement custom supplied versions of `TextOutA/W()` and `ExitProcess()` API
- Log events to a file
- The system is implemented for x86 machines running Windows 9x, Me, NT or W 2K operating system

Design and implementation

This part of the article discusses the key components of the framework and how do th interact each other. This outfit is capable to capture any kind of WINAPI imported by n functions.

Before I outline the system's design, I would like to focus your attention on several m for injecting and hooking.

First and foremost, it is necessary to select an implanting method that will meet the requirements for injecting the DLL driver into all processes. So I designed an abstract approach with two injecting techniques, each of them applied accordingly to the settin INI file and the type of the operating system (i.e. NT/2K or 9x). They are - System-w Windows Hooks and `CreateRemoteThread()` method. The sample framework offers th to inject the DLL on NT/2K by Windows Hooks as well as to implant by `CreateRemote` means. This can be determined by an option in the INI file that holds all settings of th system.

Another crucial moment is the choice of the hooking mechanism. Not surprisingly, I d apply altering IAT as an extremely robust method for Win32 API spying.

To achieve desired goals I designed a simple framework composed of the following components and files:

- `TestApp.exe` - a simple Win32 test application that just outputs a text using `Tex` API. The purpose of this app is to show how it gets hooked up.
- `HookSrv.exe` - control program
- `HookTool.DLL` - spy library implemented as Win32 DLL
- `HookTool.ini` - a configuration file
- `NTProcDrv.sys` - a tiny Windows NT/2K kernel-mode driver for monitoring proce creation and termination. This component is optional and addresses the problem detection of process execution under NT based systems only.

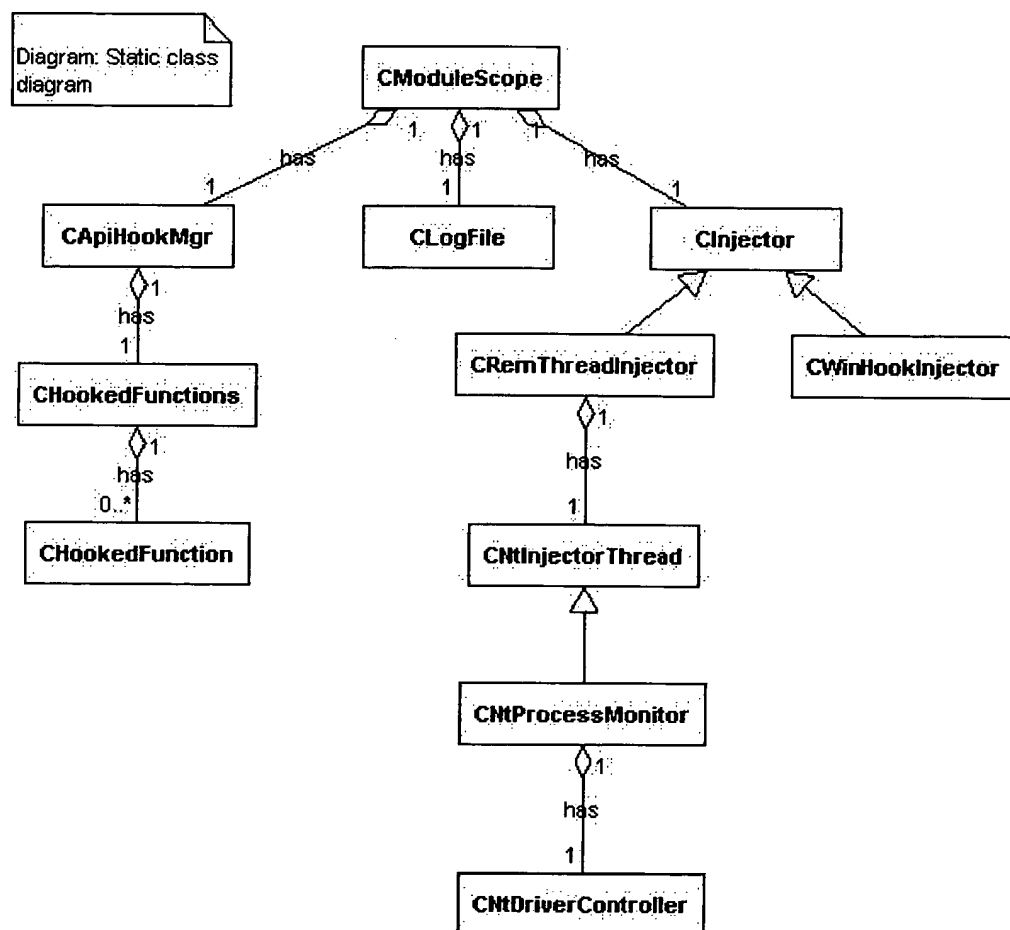
`HookSrv` is a simple control program. Its main role is to load the `HookTool.DLL` and th activate the spying engine. After loading the DLL, the Hook Server calls `InstallHook` function and passes a handle to a hidden windows where the DLL should post all mes

`HookTool.DLL` is the hook driver and the heart of presented spying system. It implem actual interceptor and provides three user supplied functions `TextOutA/W()` and `Exit` () functions.

Although the article emphasizes on Windows internals and there is no need for it to be oriented, I decided to encapsulate related activities in reusable C++ classes. This app provides more flexibility and enables the system to be extended. It also benefits developers with the ability to use individual classes outside this project.

Following UML class diagram illustrates the relationships between set of classes used HookTool.DLL's implementation.

Figure 6



In this section of the article I would like to draw your attention to the class design of HookTool.DLL. Assigning responsibilities to the classes is an important part of the development process. Each of the presented classes wraps up a specific functionality and represents a particular logical entity.

CModuleScope is the main doorway of the system. It is implemented using "Singleton" and works in a thread-safe manner. Its constructor accepts 3 pointers to the data describing the shared segment, that will be used by all processes. By this means the values of the system-wide variables can be maintained very easily inside the class, keeping the rule of encapsulation.

When an application loads the HookTool library, the DLL creates one instance of CModuleScope on receiving DLL_PROCESS_ATTACH notification. This step just initializes the only instance.

CModuleScope. An important piece of the CModuleScope object construction is the creation of an appropriate injector object. The decision which injector to use will be made after parsing the HookTool.ini file and determining the value of UseWindowsHook parameter under the [Hooks] section. In case that the system is running under Windows 9x, the value of this parameter won't be examined by the system, because Windows 9x doesn't support injecting by remote threads.

After instantiating of the main processor object, a call to ManageModuleEnlistment() will be made. Here is a simplified version of its implementation:

```
// Called on DLL_PROCESS_ATTACH DLL notification
BOOL CModuleScope::ManageModuleEnlistment()
{
    BOOL bResult = FALSE;
    // Check if it is the hook server
    if (FALSE == *m_pbHookInstalled)
    {
        // Set the flag, thus we will know that the server has been installed
        *m_pbHookInstalled = TRUE;
        // and return success error code
        bResult = TRUE;
    }
    // and any other process should be examined whether it should be
    // hooked up by the DLL
    else
    {
        bResult = m_pInjector->IsProcessForHooking(m_szProcessName);
        if (bResult)
            InitializeHookManagement();
    }
    return bResult;
}
```

The implementation of the method ManageModuleEnlistment() is straightforward and examines whether the call has been made by the Hook Server, inspecting the value m_pbHookInstalled points to. If an invocation has been initiated by the Hook Server, it sets up indirectly the flag sg_bHookInstalled to TRUE. It tells that the Hook Server started.

The next action taken by the Hook Server is to activate the engine through a single cInstallHook() DLL exported function. Actually its call is delegated to a method of CModuleScope - InstallHookMethod(). The main purpose of this function is to force for hooking processes to load or unload the HookTool.DLL.

```
// Activate/Deactivate hooking
engine BOOL CModuleScope::InstallHookMethod(BOOL bActivate, HWND hWndServ)
{
    BOOL bResult;
    if (bActivate)
    {
        *m_phwndServer = hWndServ;
        bResult = m_pInjector->InjectModuleIntoAllProcesses();
    }
    else
    {
        m_pInjector->EjectModuleFromAllProcesses();
        *m_phwndServer = NULL;
        bResult = TRUE;
    }
}
```

```

    }
    return bResult;
}

```

HookTool.DLL provides two mechanisms for self injecting into the address space of an process - one that uses Windows Hooks and another that employs injecting of DLL by CreateRemoteThread() API. The architecture of the system defines an abstract class CInjector that exposes pure virtual functions for injecting and ejecting DLL. The class CWinHookInjector and CRemThreadInjector inherit from the same base - CInjector. However they provide different realization of the pure virtual methods InjectModuleIntoAllProcesses() and EjectModuleFromAllProcesses(), defined in CInjector interface.

CWinHookInjector class implements Windows Hooks injecting mechanism. It installs function by the following call

```

// Inject the DLL into all running processes
BOOL CWinHookInjector::InjectModuleIntoAllProcesses()
{
    *sm_pHook = ::SetWindowsHookEx(
        WH_GETMESSAGE,
        (HOOKPROC) (GetMsgProc),
        ModuleFromAddress(GetMsgProc),
        0
    );
    return (NULL != *sm_pHook);
}

```

As you can see it makes a request to the system for registering WH_GETMESSAGE hook server executes this method only once. The last parameter of SetWindowsHookEx() is because GetMsgProc() is designed to operate as a system-wide hook. The callback function will be invoked by the system each time when a window is about to process a particular message. It is interesting that we have to provide a nearly dummy implementation of GetMsgProc() callback, since we don't intend to monitor the message processing. We do this implementation only in order to get free injection mechanism provided by the operating system.

After making the call to SetWindowsHookEx(), OS checks whether the DLL (i.e. HookTool.DLL that exports GetMsgProc()) has been already mapped in all GUI processes. If the DLL has not been loaded yet, Windows forces those GUI processes to map it. An interesting fact is that a system-wide hook DLL should not return FALSE in its DllMain(). That's because the operating system validates DllMain()'s return value and keeps trying to load this DLL until it finally returns TRUE.

A quite different approach is demonstrated by the CRemThreadInjector class. Here the implementation is based on injecting the DLL using remote threads. CRemThreadInjector extends the maintenance of the Windows processes by providing means for receiving notifications of process creation and termination. It holds an instance of CNTInjector class that observes the process execution. CNTInjectorThread object takes care for notifications from the kernel-mode driver. Thus each time when a process is created CNTInjectorThread::OnCreateProcess() is issued, accordingly when the process ends CNTInjectorThread::OnTerminateProcess() is automatically called. Unlike the Windows Hooks, the method that relies on remote thread, requires manual injection each time

new process is created. Monitoring process activities will provide us with a simple technique alerting when a new process starts.

`CNtDriverController` class implements a wrapper around API functions for administrative services and drivers. It is designed to handle the loading and unloading of the kernel-driver `NTProcDrv.sys`. Its implementation will be discussed later.

After a successful injection of `HookTool.DLL` into a particular process, a call to `ManageModuleEnlistment()` method is issued inside the `DllMain()`. Recall the method implementation that I described earlier. It examines the shared variable `sg_bHookIns` through the `CModuleScope`'s member `m_pbHookInstalled`. Since the server's initialization already set the value of `sg_bHookInstalled` to `TRUE`, the system checks whether this application must be hooked up and if so, it actually activates the spy engine for this process.

Turning the hacking engine on, takes place in the `CModuleScope::InitializeHookManagement()`'s implementation. The idea of this method is to install hooks for some vital function family as well as `GetProcAddress()`. By this means we can monitor loading of DLLs after the initialization process. Each time when a new DLL is about to be mapped it is necessary to fix-up its import table, thus we ensure that the system won't make any call to the captured function.

At the end of the `InitializeHookManagement()` method we provide initializations for the function we actually want to spy on.

Since the sample code demonstrates capturing of more than one user-supplied function, each must provide a single implementation for each individual hooked function. This means using this approach you cannot just change the addresses inside IAT of the different functions to point to a single "generic" interception function. The spying function needs to know which function this call comes to. It is also crucial that the signature of the interception routine must be exactly the same as the original `WINAPI` function prototype, otherwise the stack will be corrupted. For example `CModuleScope` implements three static methods `MyTextOutA()`, `MyTextOutW()` and `MyExitProcess()`. Once the `HookTool.DLL` is loaded into the address space of a process and the spying engine is activated, each time when an original `TextOutA()` is issued, `CModuleScope::MyTextOutA()` gets called instead.

Proposed design of the spying engine itself is quite efficient and offers great flexibility. However, it is suitable mostly for scenarios where the set of functions for interception is known in advance and their number is limited.

If you want to add new hooks to the system you simply declare and implement the interception function as I did with `MyTextOutA/W()` and `MyExitProcess()`. Then you register it in the way shown by `InitializeHookManagement()` implementation.

Intercepting and tracing process execution is a very useful mechanism for implementing systems that require manipulations of external processes. Notifying interested parties of the starting of a new process is a classic problem of developing process monitoring systems. System-wide hooks. The Win32 API provides a set of great libraries (`PSAPI` and `ToolH`) that allow you to enumerate processes currently running in the system. Although they are extremely powerful they don't permit you to get notifications when a new process ends up. Luckily, NT/2K provides a set of APIs, documented in Windows DDK documents "Process Structure Routines" exported by `NTOSKRNL`. One of these APIs `PsSetCreateProcessNotifyRoutine()` offers the ability to register system-wide callbacks.

function which is called by OS each time when a new process starts, exits or has been terminated. The mentioned API can be employed as a simple way to for tracking down processes simply by implementing a NT kernel-mode driver and a user mode Win32 c application. The role of the driver is to detect process execution and notify the control about these events. The implementation of the Windows process's observer NTProcDrv provides a minimal set of functionalities required for process monitoring under NT based systems. For more details see articles [11] and [15]. The code of the driver can be found in the *NTProcDrv.c* file. Since the user mode implementation installs and uninstalls the driver dynamically the currently logged-on user must have administrator privileges. Otherwise won't be able to install the driver and it will disturb the process of monitoring. A way to manually install the driver as an administrator or run HookSrv.exe using offered by 2K "Run as different user" option.

Last but not least, the provided tools can be administered by simply changing the settings in an INI file (i.e. *HookTool.ini*). This file determines whether to use Windows hooks (for NT/2K) or `CreateRemoteThread()` (only under NT/2K) for injecting. It also offers a way to specify which process must be hooked up and which shouldn't be intercepted. If you want to monitor the process there is an option (Enabled) under section [Trace] that allows monitoring system activities. This option allows you to report rich error information using the methods exposed by CLogFile class. In fact CLogFile provides thread-safe implementation and you have to take care about synchronization issues related to accessing shared system resources (i.e. the log file). For more details see CLogFile and content of HookTool.ini file.

Sample code

The project compiles with VC6++ SP4 and requires Platform SDK. In a production Windows environment you need to provide PSAPI.DLL in order to use provided CTaskManager implementation.

Before you run the sample code make sure that all the settings in HookTool.ini file have been set according to your specific needs.

For those that will like the lower-level stuff and are interested in further development of kernel-mode driver NTProcDrv code, they must install Windows DDK.

Out of the scope

For the sake of simplicity these are some of the subjects I intentionally left out of this article:

- Monitoring Native API calls
- A driver for monitoring process execution on Windows 9x systems.
- UNICODE support, although you can still hook UNICODE imported APIs

Conclusion

This article by far doesn't provide a complete guide for the unlimited API hooking subject without any doubt it misses some details. However I tried to fit in this few pages just important information that might help those who are interested in user mode Win32 API spying.

References

- [1] "Windows 95 System Programming Secrets", Matt Pietrek
- [2] "Programming Application for MS Windows" , Jeffrey Richter
- [3] "Windows NT System-Call Hooking" , Mark Russinovich and Bryce Cogswell, Dr.Do Journal January 1997
- [4] "Debugging applications" , John Robbins
- [5] "Undocumented Windows 2000 Secrets" , Sven Schreiber
- [6] "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format" by M Pietrek, March 1994
- [7] MSDN Knowledge base Q197571
- [8] PView Version 0.67 , Wayne J. Radburn
- [9] "Load Your 32-bit DLL into Another Process's Address Space Using INJLIB" MSJ M
- [10] "Programming Windows Security" , Keith Brown
- [11] "Detecting Windows NT/2K process execution" Ivo Ivanov, 2002
- [12] "Detours" Galen Hunt and Doug Brubacher
- [13a] "An In-Depth Look into the Win32 PE file format" , part 1, Matt Pietrek, MSJ Fe 2002
- [13b] "An In-Depth Look into the Win32 PE file format" , part 2, Matt Pietrek, MSJ Ma
- [14] "Inside MS Windows 2000 Third Edition" , David Solomon and Mark Russinovich
- [15] "Nerditorium", James Finnegan, MSJ January 1999
- [16] "Single interface for enumerating processes and modules under NT and Win9x/2 Ivanov, 2001
- [17] "Undocumented Windows NT" , Prasad Dabak, Sandeep Phadke and Milind Borat
- [18] Platform SDK: Windows User Interface, Hooks

History

21 Apr 2002 - updated source code

12 May 2002 - updated source code

4 Sep 2002 - updated source code

3 Dec 2002 - updated source code and demo

About Ivo Ivanov



Ivo is MCP(.NET) and for the last 3 years he has been working as a Windows/C++/.NET/C# software developer for an Australian company, based He is originally from Bulgaria where he was born and where he started progra Turbo Pascal and Assembler in the early DOS days. He's spent several years working for a few great large software companies in where he had perfect opportunity to work with some of the best worldwide Wi developers. He has done a great deal of Windows internals development as well as design implementation of complex object oriented systems. His favorite language is C recently he is seriously writing articles for .NET environment. For more details <http://www.ivosoft.com>

[Click here to view Ivo Ivanov's online profile.](#)

Other popular System articles:

- Restarting the web server from your program
A small and easy to use windows service utility
- Serial library for C++

- A high-performance, complete and compact serial library for C++
- Using MC.exe, message resources and the NT event log in your own projects
A tutorial that shows how to integrate mc.exe in the build environment of VisualStudio and use it for logging and string resources
 - CSerialPort v1.03 - Serial Port Wrapper
A freeware MFC class for Win32 serial ports.

[Top]

Sign in to vote for this article: Poor ○ ○ ○ ○ ○ Excellence



Premium Sponsor



FAQ

Noise level

Medium



Search comments

Set Options

View Dynamic

Per page 25



New thread

Msgs 1 to 25 of 313 (Total: 313) (Refresh)

First Prev Next Last

Subject

Author

Date

message hooking **NEW**

percyvimal

2:26 31 Oct '03

Re: message hooking **NEW**

Ivo Ivanov

7:21 31 Oct '03

我看了 **NEW**

Ivsha

11:25 26 Oct '03

Simple method of testing race condition

vawksel

17:44 17 Oct '03

More information on race condition.

vawksel

17:24 17 Oct '03

Re: More information on race condition.

vawksel

17:29 17 Oct '03

Unhooking from process timing problem

vawksel

16:28 17 Oct '03

Screen capture hook?

Miguel Lopes

13:49 15 Oct '03

Closing HookSrv from another program

davidmontgomery

23:06 29 Sep '03

Re: Closing HookSrv from another program

vawksel

18:26 17 Oct '03

doesn't work with all threads ???

ozzer

13:12 22 Sep '03

Re: doesn't work with all threads ???

Ivo Ivanov

17:46 23 Sep '03

unicode

rosherman2

8:41 21 Sep '03

How to intercept functions loaded in runtime

Jun-Hua Li

22:57 10 Sep '03

Where is SetWindowLongPtr() with GWLP_WNDPROC being used?

Anonymous

16:24 1 Sep '03

Doesn't work

Anonymous

21:25 30 Aug '03

Re: Doesn't work

HotFox

11:00 23 Sep '03

Re: Doesn't work

Ivo Ivanov

17:41 23 Sep '03

Re: Doesn't work

HotFox

6:40 24 Sep '03

How to compile current solution under VC6

Ivo Ivanov

8:32 24 Sep '03

Re: How to compile current solution under VC6

Anonymous

22:41 24 Sep '03

Re: How to compile current solution under VC6

Anonymous

22:46 24 Sep '03

Keyboard Handler

matthew kelly

15:16 20 Aug '03

VB.Net

tvinci

21:55 18 Aug '03

Last Visit: 7:54 Wednesday 22nd October, 2003

[First](#) [Prev](#) [Next](#) [Last](#)

All Topics, MFC / C++ >> System >> General
Updated: 3 Dec 2002
Editor: Chris Maunder

Article content copyright Ivo Ivanov, 2002
everything else Copyright © CodeProject, 1999-2003.
[Advertise on The Code Project](#) | [Privacy](#)

[MSDN Communities](#) | [ASPAlliance](#) • [DevelopersDex](#) • [DevGuru](#) • [Programmers Heaven](#) • [SitePoint](#) • [Tek-Tips Forums](#) • [TopXML](#) •
[VisualBuilder](#) • [XMLPitstop](#) • [ZVON](#) • [Search Us!](#)

[> home](#) [> about](#) [> feedback](#) [> login](#)

US Patent & Trademark Office







Try the *new* Portal design
Give us your opinion after using it.








Search Results

Search Results for: **[hook<AND>((spy))]**Found **15** of **122,228** searched.

Search within Results

  [> Advanced Search](#)[> Search Help/Tips](#)Sort by: [Title](#) [Publication](#) [Publication Date](#) [Score](#)  [Binder](#)Results 1 - 15 of 15 [short listing](#)

- 1 [Groupware infrastructure: Transparent sharing and interoperation of heterogeneous single-user applications](#) 80%
 Du Li , Rui Li
Proceedings of the 2002 ACM conference on Computer supported cooperative work
November 2002
Multi-user applications generally lag behind in features or compatibility with single-user applications. As a result, users are often not motivated to abandon their favorite single-user applications for groupware features that are less frequently used. A well-accepted approach, *collaboration transparency*, is able to convert off-the-shelf single-user applications into groupware without modifying the source code. However, existing systems have been largely striving to develop generic applic ...
- 2 [Intelligent file hoarding for mobile computers](#) 80%
 Carl Tait , Hui Lei , Swarup Acharya , Henry Chang
Proceedings of the 1st annual international conference on Mobile computing and networking December 1995
- 3 [Secure virtual enclaves: Supporting coalition use of distributed application technologies](#) 77%
 **ACM Transactions on Information and System Security (TISSEC)** May 2001
Volume 4 Issue 2
The Secure Virtual Enclaves (SVE) collaboration infrastructure allows multiple organizations to share their distributed application objects, while respecting organizational autonomy over local resources. The infrastructure is transparent to applications, which may be accessed via a web server, or may be based on Java or Microsoft's DCOM. The SVE infrastructure is implemented in middleware, with no modifications to COTS operating systems or network protocols. The system enables dynamic updates to ...
- 4 [Network Protocols](#) 77%

-  Andrew S. Tanenbaum
ACM Computing Surveys (CSUR) December 1981
Volume 13 Issue 4
- 5 Ban the book?: interactive documentation and the writer's responsibility for the 77%
 human/machine interface
Liora Alschuler , Debra Schneider
Proceedings of the 6th annual international conference on Systems documentation
October 1988
- 6 A task-based architecture for application-aware adjuncts 77%
 Robert Farrell , Peter Fairweather , Eric Breimer
Proceedings of the 5th international conference on Intelligent user interfaces January 2000
Users of complex applications need advice, assistance, and feedback while they work. We are experimenting with "adjunct" user agents that are aware of the history of interaction surrounding the accomplishment of a task. This paper describes an architectural framework for constructing these agents. Using this framework, we have implemented a critiquing system that can give task-oriented critiques to trainees while they use operating system tools and software applications. Our app ...
- 7 "Lector in rebus": the role of the reader and the characteristics of 77%
 hyperreading
Licia Calvi
Proceedings of the tenth ACM Conference on Hypertext and hypermedia : returning to our diverse roots: returning to our diverse roots February 1999
- 8 Data security for Web-based CAD 77%
 Scott Hauck , Stephen Knol
Proceedings of the 35th annual conference on Design automation conference May 1998
Internet-based computing has significant potential for improving most high-performance computing, including VLSI CAD. In this paper we consider the ramifications of the Internet on electronics design, and develop two models for Web-based CAD. We also investigate the security of these systems, and propose methods for protection against threats both from unrelated users, as well as from the CAD tools and tool developers themselves. These techniques provide methods for hiding unnecessary information ...
- 9 News track: News track 77%
 Robert Fox
Communications of the ACM May 2002
Volume 45 Issue 5
- 10 Illustrative risks to the public in the use of computer systems and related technology 77%
 Peter G. Neumann
ACM SIGSOFT Software Engineering Notes January 1996
Volume 21 Issue 1

- 11 The information age and the printing press: looking backward to see ahead 77%
James A. Dewar
Ubiquity August 2000
Volume 1 Issue 25
- 12 Toward Greater Portability: A Quixotic View 77%
Graydon Ekdahl
Linux Journal May 1998
- 13 News track 77%
Rosalie Steier
Communications of the ACM April 1989
Volume 32 Issue 4
- 14 News track 77%
Robert Fox
Communications of the ACM July 1999
Volume 42 Issue 7
- 15 An overview of portable GUI software 77%
Wade Guthrie
ACM SIGCHI Bulletin January 1995
Volume 27 Issue 1
This article attempts to bring together as much information as possible concerning platform-independent Graphical User Interface (PIGUT) development kits. It is based on a FAQ list (answers to Frequently Answered Questions) maintained and periodically updated as a service to the net by the author. What is presented here is a number of tables summarizing available PIGUT's, followed by descriptions of the individual products, with reviews and users' comments where possible.

Results 1 - 15 of 15 [short listing](#)

The ACM Portal is published by the Association for Computing Machinery. Copyright © 2003 ACM, Inc.



Find:

[Documents](#)

[Citations](#)

Searching for **spying and hooking**.

Restrict to: [Header](#) [Title](#) Order by: [Citations](#) [Hubs](#) [Usage](#) [Date](#) Try: [Amazon](#) [B&N](#) [Google \(RI\)](#) [Google \(Web\)](#) [CSB](#) [DBLP](#)

No documents match Boolean query. Trying non-Boolean relevance query.

308 documents found. Only retrieving 250 documents (System busy - maximum reduced). Order: relevance to query.

[Connected Components on Distributed Memory Machines - Krishnamurthy, Lumetta.. \(1994\) \(Correct\) \(12 citations\)](#)

uses two basic operations: pointer jumping and **hooking** operation. The algorithm maintains a forest of take the form: Parent(v) Parent(Parent(v)) The **hooking** operation **hooks** a star in the forest to another / Parent(Parent(v)) The **hooking** operation **hooks** a star in the forest to another tree in the
http.cs.berkeley.edu/projects/parallel/castle/multipol/papers/conncomp.ps

[Intelligent File Hoarding for Mobile Computers - Carl Tait \(1995\) \(Correct\) \(20 citations\)](#)

on some combination of explicit hoard profiles and **spying** on a user's file accesses. Neither of these solution might be called transparent analytical **spying**. Instead of simply recording a list of file are necessary. File accesses are traced through **hooks** in the file operation router of MFS. Except for
www.bell-labs.com/user/acharya/papers/mcn95.ps.gz

[Spying as an Object-Oriented Programming Paradigm - Pachet, Wolinski, al. \(1993\) \(Correct\) \(1 citation\)](#)

Spying as an Object-Oriented Programming Paradigm

giroux@teluq.quebec.ca Abstract We introduce **spying**, a novel way of programming with objects, based
ftp.lip6.fr/lip6/softs/NeOpus/Papers/EpiTalk/SpyingTools95.ps.Z

[A Comparison of Parallel Algorithms for Connected Components - Greiner \(1994\) \(Correct\) \(8 citations\)](#)

node's parent. There are two basic operations, **hooking** and shortcutting on trees, as diagrammed in shortcutting on trees, as diagrammed in Figure 2. **Hooking** combines pairs of trees to form larger trees if When shortcutting is performed often enough and **hooking** is done as to avoid cycles, the algorithms
www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/concomp-spaa94.ps.Z

[Super Monaco: Its Portable and Efficient Parallel Runtime System - Larson Massey \(1995\) \(Correct\) \(2 citations\)](#)

scheme and a novel variable binding and **hooking** mechanism. The result of this organization is a conservative goal counting. 2) A new mechanism for **hooking** suspended goals to variables. 3) A specialized compact use of memory, in conjunction with a novel **hooking** scheme, which maintains references to suspended
ftp.cirl.uoregon.edu/pub/users/bart/papers/sm-runtime.ps.gz

[Studies for the ATLAS Second Level Trigger using Data-Strobe Link .. - Madsen \(Correct\)](#)

.45 4.2 General Principles behind **Spying** and Verifying .48 5

. 48 5 CONTENTS 6 4.3 **Spying** on a DS-Link Network .

atlasinfo.cern.ch/Atlas/documentation/thesis/madsenn/final_thesis.ps.gz

[A Parallel Algorithm for Connected Components On Distributed.. - Bus, Tvrdik \(2001\) \(Correct\)](#)

optimization is based on replacing the conditional **hooking** by rules for reducing nontrivial cycles during optimization consists in replacing conditional **hooking** with rules for reducing nontrivial cycles not distinguish between local and remote edges. **Hooking** and contraction go across boundaries of
sci.felk.cvut.cz/~xbus/Publikace/EuroPVMMP2001.ps.gz

[Time Limited Blackbox Security: Protecting Mobile Agents From.. - Hohl \(1998\) \(Correct\) \(27 citations\)](#)

hosts, we can identify the following attacks: 1. **spying** out code 2. **spying** out data 3. **spying** out control the following attacks: 1. **spying** out code 2. **spying** out data 3. **spying** out control flow 4.

www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/vignabuch.ps.gz

[Coordinating Agents with Secure Spaces - Vitek, Bryce, Oriol \(1999\) \(Correct\) \(2 citations\)](#)

Linda primitives can just as easily be used for **spying** and spoofing on other applications using the same hi y@a :P Delta j Q Q can not interfere with or **spy** on the communication, and P will always get t.
cuiwww.unige.ch/~bryce/JourSecOS.ps

The Security Architecture of the M&M Mobile Agent Framework - Marques, Santos, Silva.. (2001) (Correct)
 resource stealing, denial-of-service attacks, data **spying** and many other problems 2,3 During the last
 to the same authority and they will not attack or **spy** agents since there is no interest in it. The
www.dei.uc.pt/~pmarques/papers/pmarques2001c_itcom2001.pdf

Physical Media Independence: System Support for Dynamically.. - Jon Inouye (1997) (Correct)
 Coda [24] and OS/2 [26] use a combination of **spying** and user-level directives to build profiles of
www.cse.ogi.edu/pub/tech-reports/1997/97-001.ps.gz

Design, Implementation, and Experimentation on - Mobile Agent Security (Correct)
 malicious hosts [2] can be identified: ffl **Spying** out and manipulation of code# ffl **Spying** out and
 ffl **Spying** out and manipulation of code# ffl **Spying** out and manipulation of data# ffl **Spying** out and
www.cse.cuhk.edu.hk/~lyu/paper_pdf/569p.pdf

Enforcing Primary Key Requirements In Multilevel Relations - Sushil Jajodia (1991) (Correct)
 TC Enterprise U Exploration U Talos U U Voyager S **Spying** S Mars S S Figure 1: SOD S Starship Objective
 create a signaling channel, both tuples (Voyager, **Spying**, Mars) and (Voyager, Exploration, Talos) are
www.list.gmu.edu/confrnc/radc/ps_ver/radc91p.ps

European Science Notes Information - Bulletin Volume Pp (Correct)
 issues, providing basic engineering knowledge and **hooking** it to design tools, managing large teams,
 library functions like "motor" or "bearing" and **hook** them together into systems. These systems can be
 was developed to generate these regions and **hook** them together. An example statement in this
web.mit.edu/ctpid/www/Whitney/Europe/final-e.pdf

Challenge Problems for Separation of Concerns - Aldrich (2000) (Correct)
 by choosing a customized subset of objects and **hooking** them together to accomplish the desired
tresp.cs.utwente.nl/Workshops/OOPSLA2000/papers/aldrich.pdf

Semantic Foundations for Composition and Interoperation of.. - McCarthy, Talcott (1996) (Correct)
 with automated reasoning systems and **hooking** up diverse theorem provers. 4 3 Technical
www-formal.stanford.edu/clt/InterOp/96nsfarpa.ps

How Conceptual Leaps in - Understanding The Nature (Correct)
 electrons are already throughout the wire and **hooking** the wire up to a battery causes flow-the excess
 it has electrons and protons all along it. Once you **hook** the battery and bulb up completing the circuit,
 there are atoms along the wire and as soon as you **hook** it up, it begins to flow. The flow makes it light
www.pz.harvard.edu/Pls/../WhatsNew/..Research/area/elecaera.pdf

First 20 documents [Next 20](#)

Try your query at: [Amazon](#) [Barnes & Noble](#) [Google \(RI\)](#) [Google \(Web\)](#) [CSB](#) [DBLP](#)

CiteSeer - citeseer.org - [Terms of Service](#) - [Privacy Policy](#) - Copyright © 1997-2002 NEC Research Institute

Searching for **fly compile and ncrypt**.

Restrict to: [Header](#) [Title](#) Order by: [Citations](#) [Hubs](#) [Usage](#) [Date](#) Try: [Amazon](#) [B&N](#) [Google \(RI\)](#) [Google \(Web\)](#) [CSB](#) [DBLP](#)

No documents match Boolean query. Trying non-Boolean relevance query.

1000 documents found. **Only retrieving 250 documents (System busy - maximum reduced).** Retrieving documents... **Order: relevance to query.**

[Defective Sign Encrypt in S/MIME, PKCS7, MOSS, PEM, PGP, and XML - Davis \(Correct\)](#)

mail servers to securely annotate messages on-the-fly (hop-by-hop) primarily for the benefit of other

[Defective Sign &Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML Don](#)

verify. Russian proverb Abstract Simple Sign &Encrypt, by itself, is not very secure. Cryptographers world.std.com/~dtd/sign_encrypt/sign_encrypt7.ps

[Distributed Encryption and Decryption Algorithms - Postma, de Boer, Helme, Smit \(1996\) \(Correct\)](#)

- 1 -Distributed **Encryption** and Decryption Algorithms Andr Postma,

Abstract In this paper, we describe distributed **encryption** and decryption algorithms. These algorithms

data storage system in which data is stored in an **encrypted** form resilient to a number of arbitrarily

[www.pegasus.esprit.ec.org/people/arne/publications/encdecalg.ps](#)

[The order of encryption and authentication for protecting.. - Krawczyk \(2001\) \(Correct\) \(8 citations\)](#)

basic cryptographic notions. In this section we **compile** and highlight some of these issues. 6.1 The

The order of **encryption** and authentication for protecting

question of how to generically compose symmetric **encryption** and authentication when building \secure [eprint.iacr.org/2001/045.ps.gz](#)

[. History of DES - Note Much Of \(Correct\)](#)

1 CS 270 Lecture 20: Data **Encryption** Standard (DES) Professor Christos

bits) went on to be adopted in 1976 as the Data **Encryption** Standard (DES) Over the years, DES has been

workstations take at most a few seconds per MB of **encrypted** data. Special purpose DES chips have been

[www.cs.berkeley.edu/~christos/cs270/lec20.ps](#)

[Scramble All, Encrypt Small - Jakobsson, Stern, Yung \(1999\) \(Correct\)](#)

Scramble All, **Encrypt** Small Markus Jakobsson Julien P. Stern

paper, we propose a new design tool for \block **encryption** "allowing the en/decryption of arbitrarily

natural use of our scheme is for remotely keyed **encryption**. We actually solve an open problem (at least

[www.bell-labs.com/user/markusj/scramble.ps](#)

[MPEG Video Encryption in Real-time Using Secret Key.. - Changgui Shi Sheng-Yih \(1999\) \(Correct\) \(2 citations\)](#)

MPEG Video **Encryption** in Real-time Using Secret Key Cryptography

47906, USA. Abstract We present a fast MPEG video **encryption** algorithm called RVEA which **encrypts** selected

MPEG video **encryption** algorithm called RVEA which **encrypts** selected sign bits of the DCT coefficients and

[www.cs.purdue.edu/homes/bb/security99.ps](#)

[Performance Comparison of the AES Submissions - Schneier, Kelsey, Whiting.. \(1999\) \(Correct\) \(15 citations\)](#)

the algorithms have no way to compute subkeys on the fly, thus requiring that all the subkeys be

on current Intel CPUs, Twofish's implementation **compiles** key-dependent data directly in the **encryption**

The principal goal guiding the design of any **encryption** algorithm must be security. In the real

[www.counterpane.com/aes-performance.pdf](#)

[How to Encrypt Long Messages without Large Size.. - Mitomo, Kurosawa \(2000\) \(Correct\)](#)

How to **Encrypt** Long Messages without Large Size

Messages without Large Size Symmetric/Asymmetric **Encryption** Schemes Masashi Mitomo Kaoru Kurosawa

Abstract. Suppose that we wish to **encrypt** long messages with small overhead by a public

[eprint.iacr.org/2000/065.ps.gz](#)

[Online Aggregation - Hellerstein, Haas, Wang \(1997\) \(Correct\) \(73 citations\)](#)

aggregation queries and control execution on the fly. After outlining usability and performance

db.cs.berkeley.edu/papers/sigmod97-online.ps.Z

Selective Encryption and Watermarking of MPEG Video (Extended.. - Wu, Wu (1997) (Correct) (2 citations)
Selective **Encryption** and Watermarking of MPEG Video Extended
 a research prototype which integrates selective **encryption** schemes with watermarking techniques in one
 We have experimented 7 different selective **encryption** schemes and one selective watermarking scheme
shang.csc.ncsu.edu/papers/smpeg.ps.gz

A Highly Safe Self-Stabilizing Mutual Exclusion Algorithm - Yen, Bastani (1996) (Correct) (7 citations)
 The top processor has knowledge of a private **encryption** key K T while the bottom processor has
 the bottom processor has knowledge of a private **encryption** key KB .The public keys (for decryption)D
 processor: do S =decrypt(D B R) CR S :**encrypt**(K T R) end do Other processors: do S =
www.utdallas.edu/~ilyen/papers/ipl.ps

Customized Dynamic Load Balancing for a Network of Workstations - Mohammed Javeed (1995) (Correct) (4 citations)
 for good performance. We present a hybrid **compile**-time and run-time modeling and decision process
ftp.cs.rochester.edu/pub/papers/systems/96.HPDC.Customized_dynamic_load_balancing.ps.gz

Optimized Software Synthesis for Digital Signal.. - Jürgen Teich.. (1998) (Correct) (1 citation)
 and consumed by each actor is fixed and known at **compile**-time. Example 1.1 Figure 1.1 shows a simple SDF
ftp.tik.ee.ethz.ch/pub/people/zitzler/TZB1998a.ps.gz

Optimizing ML with Run-Time Code Generation - Leone, Lee (1995) (Correct) (91 citations)
 [14] David Keppel. A portable interface for on-the-fly instruction space modification. In Proceedings of
 We describe the design and implementation of a **compiler** that automatically translates ordinary programs
foxnet.cs.cmu.edu/~petel/papers/staged/mleone-pldi96.ps

Time-lock puzzles and timed-release Crypto - Rivest, Shamir, Wagner (1996) (Correct) (20 citations)
 of "timed-release crypto, where the goal is to **encrypt** a message so that it can not be decrypted by
 wants to give his mortgage holder a series of **encrypted** mortgage payments. These might be **encrypted**
 of **encrypted** mortgage payments. These might be **encrypted** digital cash with different decryption dates,
now.cs.berkeley.edu/~daw/timelock.ps

Storing Classified Files - Halevi, Petrank (1995) (Correct)
 sensitive information they need to be stored **encrypted** in the system. Once **encryption** strategy is
 need to be stored **encrypted** in the system. Once **encryption** strategy is set, the problem of key
 of key distribution arises. That is, we need to **encrypt** the files using some keys and then somehow
theory.lcs.mit.edu/pub/people/shaih/classify.ps.gz

ADAPTOR Users Guide Version 6.0 - Brandes, Höver-Klier (1998) (Correct)
 . 10 5.12 **Compiler** Optimization .
unix.hensa.ac.uk/parallel/languages/fortran/adaptor/docs/uguide_6.0.ps

Towards a Crystal Ball for Data Retrieval - Hellerstein (Correct)
 results, control the behavior of the queries on the fly, and better understand the operation of the
s2k-ftp.cs.berkeley.edu/postgres/papers/ngits97-control.ps.Z

dSPACE Software Environment in the ARO-DURIP Facility - Koutsoukos (1997) (Correct)
 a host program to alter system parameters on-the-fly, i.e. while the DSP is executing the DSP program.
 ffl Register the DSP boards at the daemon. ffl **Compile** and download the file to the DSP controller.
www.nd.edu/~isis/techreports/isis-97-003.ps.Z

First 20 documents [Next 20](#)

Try your query at: [Amazon](#) [Barnes & Noble](#) [Google \(RI\)](#) [Google \(Web\)](#) [CSB](#) [DBLP](#)

CiteSeer - citeseer.org - [Terms of Service](#) - [Privacy Policy](#) - Copyright © 1997-2002 [NEC Research Institute](#)

Your Search:

[Advanced Web Search](#)
[Preferences](#)

[Web](#)

[Images](#)

[Directory](#)

[Yell w Pages](#)

[News](#)

[Products](#)



TOP 20 WEB RESULTS out of about 2,800

1. [API Spying Techniques for Windows 9x, NT and 2000](#)
API Spying Techniques for Windows 9x, NT and 2000 Yariv Kaplan API spying utilities ...
What all of this got to do with API spying? Well, it seems ...
www.internals.com/articles/apispy/apispy.htm - 32k - [Cached](#) - [More pages from this site](#)
2. [Softlookup.com - API Spy 32-Display Information](#)
... Description: API Spy 32 API Spy 32 allows you to examine any known API function calls that are ... The results of a spying session can be saved in a log file. ...
www.softlookup.com/preview/dis7081.html - 8k - [Cached](#)
3. [vbCity.com Forums » VOSP » VOSP General » API Spy](#)
... What do you thing about VB ApiSpy sub? I'm ready to share existing code (it's just an idea - **spying** 1 app using GetThreadContext API). ...
www.devcity.net/forums/topic.asp?tid=8024 - 85k - [Cached](#)
4. [sumbera.com:MicroStation:research:MDL API Spy](#)
... so how it works now ? you need only 2 lines to get into API function : ... here is example of **spying** of mdlDialog_callFunction with print of calling MDL task. ...
www.sumbera.com/ustation/research/mdlspy.htm - 57k - [Cached](#)
5. [APISpy32 free download. A system-wide API spying utility capable ...](#)
A system-wide API spying utility capable of intercepting API calls made by all active Windows processes and their attached DLLs. It gathers informatio....
www.freedomdownloadcenter.com/Utilities/System_Analysis_Uutilities/APISpy32.html - 17k - [Cached](#) - [More pages from this site](#)
6. [api spy](#)
... METHOD. DETAIL: FIELD| CONSTR| METHOD. API Spying Techniques for Windows 9x, NT and 2000 Yariv Kaplan API API spying utility. First ...
www.lolathemovie.com/william-novak.htm
7. [Spying in Windows Graphics System](#)
User interface for Pogy: API spying program. API spying through hooking imported API functions. API spying through hooking all API ...
www.fengyuan.com/sample/samplech4.html - 2k - [Cached](#)
8. [loadsoft.narod.ru - Debugging, Tracing, Monitoring and Spying. ...](#)
... new releases, top downloads, editors' picks: API Spy 32 v2.5 update is a tool for examining API functions used by 32-bit Windows applications. ...
loadsoft.narod.ru/programming/debugging_and_tracing/ - 20k - [Cached](#)
9. [Frequently Asked Questions](#)
... A major functional overhaul to APISPY32 isn't planned. BoundsChecker and other API spy tools already do a much better job of API spying. Here's one such tool. ...
www.wheaty.net/FAQ.htm - 11k - [Cached](#)
10. [The Code Project - Using GINA.DLL to Spy on Windows User Name & ...](#)



... profile. Other popular System articles: **API hooking revealed** The article demonstrates how to build a user mode **Win32 API spying** system; ...
www.codeproject.com/useritems/GINA_SPY.asp - 46k - [Cached](#) - [More pages from this site](#)

11. [Auto Debug for Windows - Auto spy all API and COM Interface. ...](#)
... Supporting **spying** the parameters of function before and after the function is called. ...
You can trace and monitor **API** without programming. ...
www.soft14.com/Software_Development/Compilers_Interpreters_and_Debugging/Auto_Debug_for_Windows_270_Review.html - 12k - [Cached](#)
12. [window programming api control](#)
... You should always use the APIs, rather **API Spying** Techniques for Windows 9x, NT and 2000 Yariv Kaplan **API spying** utilities are among the most powerful tools ...
www.nisinvotec.com/telemarketing-job-at-home.htm
13. [Programmers Heaven - Windows Zone - Homepages Links](#)
... Advertisement. Matt Pietreks' site Some really good stuff like **API Spying**, Matt's articles etc. The page on **API Spy** is especially nice.....
www.programmersheaven.com/zone15/links/link327.htm - 38k - [Cached](#) - [More pages from this site](#)
14. [VN Informatics::Tutorial - Các bí mậ cấa việ API Hook ...](#)
... hook.Việ dùng cái nào thì tùy vào việ bậ muậ làm trong trườg hậ cậ thậ General design of an **API spying** framework : Luôn luôn ...
www.vninformatics.com/portal/NewsTopic/Tutorial/1023433432/ - 54k - [Cached](#)
15. [reverse-engineering](#)
... 9x/NT **API** Hooking via Import Tables *New. **API Spying** Techniques for Windows 9x, NT and 2000. Hooking Windows NT System Services. Hooking Software Interrupts.
www.yates2k.net/syshook.html - 9k - [Cached](#)
16. [ForwardLab, Inc.](#)
... The CeRegSpy is based on CE **API spying** techniques presented in article "Spy: A Windows CE **API** interceptor" in October 2003 issue of Dr. Dobb's Journal. ...
www.forwardlab.com/ceregsy.htm - 10k - [Cached](#)
17. [API Hook Source Code](#)
... Keywords: **API** hooking, system-wide **API** Hook , interceptor, **API spying** , system-wide **API** interceptor, **API** interception library, intercept a windows **API**, hook **API** ...
www.apihook.com/apihook/index.shtml - 11k - [Cached](#)
18. [Контроль вызова API функций в среде ...](#)
... Данная статья является очень вольным переводом статьи "**API Spying** Techniques For Windows 95, 98, NT ...
www.codenet.ru/progr/other/spyapi.php - 22k - [Cached](#)
19. [How to write Game spying tools like Game Make Easy](#)
... A powerful **spying** tools may need more system related **API**, of which the following APIs need some explanation: DebugActiveProcess(). ...
home.hkstar.com/~yklui/How2Write1.htm - 17k - [Cached](#)
20. [spy program](#)
... How to Stop Spyware from **Spying** On You, ... TracePlus/Win32 is a **API** Trace/Debugger for the Microsoft, <http://www.sstinc.com/windows.html>, ...
www.security-products-links.com/spy_program.html - 76k - [Cached](#) - [More pages from this](#)

[site](#)

Results Page:

1 [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [▶ Next](#)

[Web](#)

[Images](#)

[Directory](#)

[Yellow Pages](#)

[News](#)

[Products](#) NEW!

Your Search:

[Search](#)

[Advanced Web Search](#)
[Preferences](#)

Copyright © 2003 Yahoo! Inc. All rights reserved. [Privacy Policy](#) - [Terms of Service](#) - [Ad Feedback](#)

Search Technology provided by Google

[Advanced Search](#)[Preferences](#)[Language Tools](#)[Search Tips](#)

[Web](#) · [Images](#) · [Groups](#) · [Directory](#) · [News](#)
Searched the web for **hooking spying**.

Results **1 - 10** of about **3,700**. Search took **0.50** seconds.

API Spying Techniques for Windows 9x, NT and 2000

... API **spying** utilities are among the most powerful tools for exploring the inner ... light on this subject by presenting several techniques for **hooking** API calls ...

www.internals.com/articles/apispy/apispy.htm - 32k - [Cached](#) - [Similar pages](#)

How to write Game spying tools like Game Make Easy

... The lpfn (hook procedure) must exist in a DLL if you're planning for a global hook, which is the type of **hooking** needed for the **spying** tools. ...

home.hkstar.com/~yklui/How2Write1.htm - 17k - [Cached](#) - [Similar pages](#)

Spying in Windows Graphics System

User interface for Pogy: API **spying** program. API **spying** through **hooking** imported API functions. API **spying** through **hooking** all API ...

www.fengyuan.com/sample/samplech4.html - 2k - [Cached](#) - [Similar pages](#)

The Code Project - API hooking revealed - System

... These are some of the issues addressed by the Hook Tool System: Provide a user-level

hooking system for **spying** any Win32 API functions imported by name; ...

www.codeproject.com/system/hooksys.asp - 101k - [Cached](#) - [Similar pages](#)

reverse-engineering

... VxD Tutorial Part Three(API **Hooking**). 9x/NT API **Hooking** via Import Tables *New. API **Spying** Techniques for Windows 9x, NT and 2000. ...

www.yates2k.net/syshook.html - 9k - [Cached](#) - [Similar pages](#)

Gone Fishin': Hooking the Internet Explorer 4.0 Object Model

... WithEvents does the same thing I talked about above regarding **hooking** in C++: it finds a ... **Spying** on IE 4.0 with SplE4 With the HookUp control up and ready, I ...

www.microsoft.com/mind/1297/hookie.asp - 59k - [Cached](#) - [Similar pages](#)

Subclassing & Hooking with Visual Basic: Chapter 1: Introduction

... with using these advanced techniques, effectively implementing subclassing and **hooking** in our ... Spy++ is described as a tool for "**spying**" on different parts of ...

www.oreilly.com/catalog/subhookvb/chapter/ch01.html - 40k - [Cached](#) - [Similar pages](#)

7Online.com: "Hooking" A Peeping Tom

... **Hooking** A Peeping Tom. (Wichita, Kansas-AP, September 23, 2002) — A father used fishing line to catch a man he suspected of **spying** on his daughters. ...

abclocal.go.com/wabc/features/WABC_strange_092302tom.html - 20k - [Cached](#) - [Similar pages](#)

The Code Project - Hide String value from Regedit by Hooking the ...

... profile. Other popular System articles: API **hooking** revealed The article demonstrates how to build a user mode Win32 API **spying** system; ...

codeproject.com/system/hidereg.asp - 32k - [Cached](#) - [Similar pages](#)

[[More results from codeproject.com](#)]

Sponsored Links

Secret Spying Software

Record everything they do on the Internet and computer. Affiliate.

www.key-recorder.com

Interest: [XXXXXXXXXX](#)

Free Spyware/Adware Scan

Scan your PC for Spyware & Adware infections absolutely free now.

SpywareBegone.com

Interest: [XXXXXXXXXX](#)

Spygate.com - Spy Store

Wireless Clock Radio, Pager & More. Hidden Camera Video. 800-320-4888.

www.spygate.com

Interest: [XXXXXXXXXX](#)

Spy with your Computer

Record whatever others type or read on your computer.

www.computerwatchdog.com

Interest: [XXXXXXXXXX](#)

[See your message here...](#)

Spying

... If any of you are **hooking** up there you should give me a shout 'cause chances are I'll be around (933 ... Follow Ups: Re: **Spying** Clint Gosling 19:22:08 9/18/98 (0): ... www.engr.usask.ca/~chess/mboard/messages/105.html - 4k - [Cached](#) - [Similar pages](#)

Google

Result Page: 1 2 3 4 5 6 7 8 9 10 [Next](#)

hooking spying

Google Search

[Search within results](#)

Dissatisfied with your search results? [Help us improve.](#)

Get the [Google Toolbar](#):



[Google Home](#) - [Advertise with Us](#) - [Business Solutions](#) - [Services & Tools](#) - [Jobs, Press, & Help](#)

©2003 Google



[Advanced Search](#) [Preferences](#) [Language Tools](#) [Search Tips](#)

api spying

Google Search

Web · [Images](#) · [Groups](#) · [Directory](#) · [News](#)
Searched the web for **api spying**.

Results **1 - 10** of about **4,370**. Search took **0.34** seconds.

API Spying Techniques for Windows 9x, NT and 2000

API Spying Techniques for Windows 9x, NT and 2000 Yariv Kaplan **API spying** utilities ... What all of this got to do with **API spying**? Well, it seems ...
www.internals.com/articles/apispy/apispy.htm - 32k - [Cached](#) - [Similar pages](#)

Utilities

... APISpy32 **API spying** utilities are the most powerful tools for exploring the internal structure of applications and operating systems. ...
www.internals.com/utilities/utilities.htm - 3k - [Cached](#) - [Similar pages](#)
[[More results from www.internals.com](#)]

Softlookup.com - API Spy 32-Display Information

... Description: **API Spy 32** **API Spy 32** allows you to examine any known **API** function calls that are ... The results of a **spying** session can be saved in a log file. ...
www.softlookup.com/preview/dis7081.html - 8k - [Cached](#) - [Similar pages](#)

vbCity.com Forums » VOSP » VOSP General » API Spy

... What do you thing about VB ApiSpy sub? I'm ready to share existing code (it's just an idea - **spying** 1 app using GetThreadContext **API**). ...
www.devcity.net/forums/topic.asp?tid=8024 - 85k - Nov 1, 2003 - [Cached](#) - [Similar pages](#)

sumbera.com:MicroStation:research:MDL API Spy

... so how it works now ? you need only 2 lines to get into **API** function : ... here is example
of **spying** of mdlDialog_callFunction with print of calling MDL task. ...
www.sumbera.com/ustation/research/mdlspey.htm - 57k - [Cached](#) - [Similar pages](#)

APISpy32 free download. A system-wide API spying utility capable ...

A system-wide **API spying** utility capable of intercepting **API** calls made by all active Windows processes and their attached DLLs. It gathers informatio....
www.freedownloadscenter.com/Utilities/System_Analysis_Uilities/APISpy32.html - 17k - [Cached](#) - [Similar pages](#)

api spy

... METHOD. DETAIL: FIELD| CONSTR| METHOD. **API Spying Techniques for Windows 9x, NT and 2000** Yariv Kaplan **API API spying** utility. First ...
www.lolathemovie.com/william-novak.htm - [Similar pages](#)

Spying in Windows Graphics System

User interface for Pogy: **API spying** program. **API spying** through hooking imported **API** functions. **API spying** through hooking all **API** ...
www.fengyuan.com/sample/samplech4.html - 2k - [Cached](#) - [Similar pages](#)

loadsoft.narod.ru - Debugging, Tracing, Monitoring and Spying, ...

... new releases, top downloads, editors' picks: **API Spy 32** v2.5 update is a tool for examining **API** functions used by 32-bit Windows applications. ...
loadsoft.narod.ru/programming/debugging_and_tracing/ - 20k - [Cached](#) - [Similar pages](#)

Sponsored Links

Secret Spying Software
Record everything they do on the Internet and computer. Affiliate.
www.key-recorder.com
Interest: ██████████

Free Spyware/Adware Scan
Scan your PC for Spyware & Adware infections absolutely free now.
SpywareBegone.com
Interest: ██████████

Spygate.com - Spy Store
Wireless Clock Radio, Pager & More.
Hidden Camera Video. 800-320-4888.
www.spygate.com
Interest: ██████████

Spy with your Computer
Record whatever others type or read on your computer.
www.computerwatchdog.com
Interest: ██████████

[See your message here...](#)

Frequently Asked Questions

... A major functional overhaul to APISPY32 isn't planned. BoundsChecker and other API spy tools already do a much better job of **API spying**. Here's one such tool. ...

www.wheaty.net/FAQ.htm - 11k - [Cached](#) - [Similar pages](#)

Goooooooooooooogle ►

Result Page: [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [Next](#)

api spying

Google Search

[Search within results](#)

Dissatisfied with your search results? [Help us improve.](#)

Get the [Google Toolbar](#):



[Google Home](#) - [Advertise with Us](#) - [Business Solutions](#) - [Services & Tools](#) - [Jobs, Press, & Help](#)

©2003 Google



Search for

MSDN Home > MSJ > September 1997

MSDN Magazine

Go

Advanced Search

MSJ Home

September 1997

Search

Source Code

Back Issues

Subscribe

Reader Services

Write to Us

MSDN Magazine

MIND Archive

Magazine Newsgroup

September 1997

MICROSOFT SYSTEMS JOURNAL

UNDER THE HOOD

MATT PIETREK

Code for this article: Hood0997.exe (48KB)

Matt Pietrek is the author of Windows 95 System Programming Secrets (IDG Books, 1995). He works at NuMega Technologies Inc., and can be reached at mpietrek@tiac.com or at <http://www.wheaty.net>.

It's not every day that I end up working with assembly language, API interception, the Internet, and Visual Basic® all in the same project. While API spying is up my alley, what the heck am I doing writing Visual Basic and Internet code?

One night, while sitting in the spa, it occurred to me that I wanted to more actively monitor the mutual funds in my retirement plans. There are a lot of great Web sites out there that offer free stock and mutual fund analysis and charting. My favorite is the Microsoft® Investor site (<http://investor.msn.com>), but since my personal T1 line hasn't arrived yet, it's painful waiting for my 28.8Kbps modem to grind through downloads.

Also, being a programmer, I naturally wanted the raw data so I could do my own custom analysis and charting. While I can easily use my browser to find the prices of each fund I own, it would be a real pain to manually transcribe all the prices from the browser page. Even worse, the process would need to be repeated every day. No, I wanted a program that used the Web to retrieve stock or mutual fund prices without requiring me to do anything manually. No mouse clicks, no typing. Just run the program and my personal database of fund prices is updated.

Drawing on the vast emptiness of my Internet programming knowledge, it occurred to me that, when using a browser to get an online quote, it's really just an HTTP transaction between your browser and a server somewhere. By determining what data a browser sends and receives when you submit a quote request, it's possible to create a program that mimics

these interactions, thereby receiving the same data that a browser would. Luckily, my last remaining bit of Internet knowledge was that Microsoft Internet Explorer (IE) 3.0x uses WININET.DLL—a Win32® system DLL that provides a high-level layer over the HTTP, FTP, and Gopher protocols, sparing you from the nastiness of Windows® socket programming.

By observing the calls made to WININET.DLL while retrieving a quote, it's possible to create a program that makes similar calls to the WININET APIs, but without the overhead of an entire Internet browser. Of course, the ability to watch what IE is doing is useful for many things beyond mere stock quotes. For example, I found it extremely interesting to watch how IE did things like downloading Java classes, using cookies, and caching the most recently downloaded pages and image files.

Longtime readers of *MSJ* may recall an article I wrote a while back that presented a generic API spying program called APISPY32. While theoretically I could have used APISPY32 to monitor calls made to WININET.DLL, there would have been a variety of technical hurdles that I won't go into here. Instead, I opted to use a more classical method of API spying. I called the resulting code WininetSpy.

The core of WininetSpy is a DLL that shares a common name with WININET.DLL and exports many of same functions as the Microsoft-supplied WININET.DLL. Each exported function in my DLL performs whatever logging is needed in addition to calling its corresponding API in the Microsoft WININET.DLL. Theoretically, my version should export a stub API for every API in the Microsoft WININET.DLL. But not all of the functions in WININET.DLL are documented, so it would be difficult (but not impossible) to create stubs for these undocumented APIs. In addition, I couldn't find any programs that used the Unicode version of the WININET APIs. Therefore, I was lazy and only provided logging stubs for the WININET APIs that are actually used by IE 3.0x.

Two system-supplied DLLs are layered between IE and WININET.DLL: MSHTML.DLL and URLMON.DLL. By combining the list of WININET functions imported by these two DLLs, I came up with the list of functions that my WININET.DLL had to export. Important note: the WininetSpy code was tested extensively on IE 3.02, using Windows NT® 4.0. If future versions of IE import additional WININET functions, IE will likely stop functioning if my WININET.DLL is being used. The fix would be to add stub logging functions for the appropriate new WININET APIs. Alternatively, you could just remove my WININET.DLL from its installed location, and everything should work correctly afterwards. You'll see why momentarily.

At this point, you probably have two concerns about how my WININET.DLL fits into the picture. First, aren't there restrictions about having two DLLs with the same name in a process? Second, how can I force the system to connect MSHTML.DLL and URLMON.DLL to my replacement WININET.DLL rather than the system-supplied WININET.DLL? The answer to both comes down to location, location, and location.

Unlike 16-bit Windows, Win32 doesn't care if you have multiple DLLs with the same name in a process address space. When keeping track of loaded DLLs, Win32 operating systems use the DLL's complete path as its name. This is different than 16-bit Windows, which uses the base file name of the DLL (such as WININET) as the module name. So you can have two copies of WININET.DLL loaded as long as they're in different directories. The tricky part is getting them both loaded and everything hooked up properly.

By examining the documentation for the Win32 loader (essentially, the LoadModule API), you'll see that the loader first searches for DLLs in the directory where the application's executable resides. This is perfect for what WininetSpy needs to do. By dropping my replacement WININET.DLL into the same directory as IE (IEXPLORE.EXE), I can force my WININET.DLL to be loaded rather than the Microsoft WININET.DLL (which is in the system directory). Once my WININET.DLL is loaded, it's a simple matter to call LoadLibrary to load the real WININET.DLL. Let's look at some code now to see what I've just described.

Figure 1 shows the code for WININETSPY.CPP, which compiles into WININET.DLL using the supplied makefile. Most of the code is boilerplate API stub functions that perform the logging before calling the real WININET APIs in the system's WININET.DLL. I'll describe them later. For now, concentrate on theDllMain function near the top. The code executed when the DLL loads (inside the DLL_PROCESS_ATTACH if clause) first disables thread notifications by calling the DisableThreadLibraryCalls. My WININET.DLL doesn't need to know about thread creation and termination, so this call tells the system not to bother calling my DllMain for thread-related activity. Next, DllMain attempts to load the Microsoft-supplied WININET.DLL by calling LoadLibrary with a complete path for the DLL. My code assumes that the system WININET.DLL will be in the Win32 system directory, which it locates via the GetSystemDirectory API.

If everything goes as planned, after the LoadLibrary call returns the system WININET.DLL is loaded and ready to go. The next major task is to look up the address of each of the APIs exported by the system WININET.DLL. As you might expect, I do this by calling GetProcAddress on each exported API. Since WININET.DLL exports well over 100 functions, you

might expect to see a whole bunch of GetProcAddress calls somewhere in WININETSPY.CPP. You won't find them, though. The closest thing you'll find is this:

```
#define SPYMACRO( x ) \
    g_pfn##x = GetProcAddress( hModWininet, #x );
#include "wininet_functions.inc"
```

This code fragment makes extensive use of the C++ preprocessor to automate the generation of boilerplate code. The SPYMACRO macro uses the token pasting operator (##) and the stringizing operator (#) to create a macro that takes a function name as input and expands to something like:

```
g_pfnInternetOpenA =
    GetProcAddress(hModWininet, "InternetOpenA" );
```

The line that reads #include "wininet_functions.inc" is just a list of the functions exported from WININET.DLL. The contents of this file begin like this:

```
SPYMACRO( AuthenticateUser )
SPYMACRO( CommitUrlCacheEntryA )
```

The final result of this preprocessor funny business is: for each function listed in WININET_FUNCTIONS.INC, DllMain calls GetProcAddress on that function and assigns the return address to an appropriately named function pointer declared at the global scope. Why go through all the hassle of listing the WININET APIs in a separate file? Why not just include the API list directly in DllMain? Think about all those function pointers that need to be declared at global scope, and therefore outside of the DllMain code. With over 100 WININET APIs, I'd be looking at adding 100 or so additional lines of code to declare these variables.

By putting the API list in a separate file, I can use a different definition for the SPYMACRO macro and #include the "wininet_functions.inc" file a second time. This time, SPYMACRO looks like:

```
#define SPYMACRO( x ) FARPROC g_pfn##x;
```

This expands to something like:

```
FARPROC g_pfnInternetOpenA;
```

Localizing all of the API functions in a separate file has two advantages. First, if I wanted to change the variable names, the call to GetProcAddress, or whatever, I'd make the change in exactly one spot.

That is, where the SPYMACRO is declared. Second, if I were to add new WININET API functions to the file, both the function pointer declaration and its corresponding GetProcAddress would automatically appear upon recompiling. (I've read that Bjarne Stroustrup isn't enamored of preprocessors and macros, but I think that they're pretty slick when you can do things like this.)

The remaining code in DIIMain is simple code for opening the logging output file and closing it when the process terminates. I'll get to the somewhat unusual output file later. For now let's focus on the code that makes up the majority of WININETSPY.CPP: the API logging stubs.

Take a glance at code for InternetCanonicalizeUrlA (it's the first API stub). As you'd expect, the return value, calling convention, and parameters are exactly the same as the API's prototype in WININET.H. In fact, I simply copied the relevant WININET.H prototypes into WININETSPY.CPP and made functions out of them. The meat of each API logging stub is fairly standard and goes like this:

- Declare a local variable to store the return value.
- Call the real WININET API function with the SPYCALL macro.
- Log the API's name and relevant parameters.
- Return the value that the real WININET API returned.

The most interesting part of the sequence is the SPYCALL macro. If you've ever used a function pointer returned by GetProcAddress, you know what a pain it can be. In C++, you need to make a typedef corresponding to the function definition, and typecast the return value from GetProcAddress to this typedef:

```
typedef INTERNETAPI
    (BOOL WINAPI *PFNINTERNETCLOSEHANDLE)
    (INTERNET hInternet);

g_pfnInternetCloseHandle =
    (PFNINTERNETCLOSEHANDLE)GetProcAddress(
        hModWininet, "InternetCloseHandle");
```

Yuck! What a mess! Now multiply this hassle by over 100 WININET APIs. Alas, it has to be like this so that the compiler can verify parameters and return values for their proper type. The SPYMACRO macro trades off this type safety for a much easier way to invoke the real WININET APIs. Check out the SPYMACRO code near the beginning of WININET.CPP.

The SPYMACRO macro is a sequence of inline assembler instructions that use the two macro parameters: a function pointer to be called and the

number of DWORDs passed as arguments to the API. Luckily, the number of DWORDs is usually the same as the number of arguments. The assembler code makes a copy of the API's parameters to a lower location on the stack, and then calls through the function pointer. The function pointer is what transfers control to the real API code in the system-supplied WININET.DLL. Looking through the code, you'll see that the function pointer parameter to SPYMACRO is always one of the g_pfnXXX global variables that I described earlier.

After the real API code returns, the SPYCALL macro cleans the copied parameters off the stack and copies the return value (in EAX) to a local variable. The SPYMACRO code assumes that you've declared a local variable named retValue, a small price to pay for ridding yourself of the compiler's obsessive type checking. All in all, this dancing on the fringe isn't recommended programming practice, but if you're experienced enough to understand the risks and rewards, I say go for it!

After the SPYCALL macro code executes, the logging code in each API stub comes next. To be honest, I didn't bother to log every parameter of every API. Rather, I selected the parameters that were most likely to be informative, strings in particular. Feel free to add to the list of parameters that it logs. To handle the cases where a string parameter might be zero, I wrote the SAFESTR macro. For a given input string, it returns either the same pointer, or a pointer to an empty string ("") if the input string pointer is zero. This let me avoid cluttering up the code with hundreds of checks for valid string pointers.

The last piece of the WININETSPY.CPP to look at is the actual logging code. While the logging occurs via a function called printf, this printf isn't the standard C++ runtime library version of the function. I wrote a replacement printf function (near the top of WININETSPY.CPP) that formats the output like printf would and writes the results to a file. My replacement printf assumes that a global variable named g_hOutputFile has been initialized with a valid file handle. DIIMain is where this initialization occurs.

When I first wrote WININETSPY.CPP, my output file was an ordinary disk-based text file. Plain, simple, and easy to work with in an editor. Eventually, I became dissatisfied with disk files. I wanted to see the logging trace as it occurred, and I didn't want to hassle with opening up the output file in an editor. I also wanted to be able to easily throw away all prior logging output and start with a fresh, clean buffer. For example, in figuring out the operations needed to get a fund quote, there are hundreds of uninteresting output lines emitted before getting to the point where you'd click on the Get Quote button. In short, I wanted the logging output to be collected and presented in a

different program.

After pondering possible implementations, I hit upon the idea of using the Win32 mailslot facility. Instead of opening a disk file in DllMain, I instead open an existing mailslot and write to its file handle. Nothing else needs to change. I don't have space here to describe mailslots in any detail. The important thing is that I could treat each line of logging output as a message and lob it into the mailslot. The logging display program simply needs to read from the mailslot in a timely manner and display each message.

At this point, Visual Basic entered the picture. In a manner of minutes, I whipped together a Visual Basic program consisting of a form and an edit control where I appended each line of output read from the mailslot. Later, I got fancy and changed the edit control to a rich text control to get cool features like searching and buffers greater than 64KB. I called the finished program WININETSPYMon (see **Figure 2**).

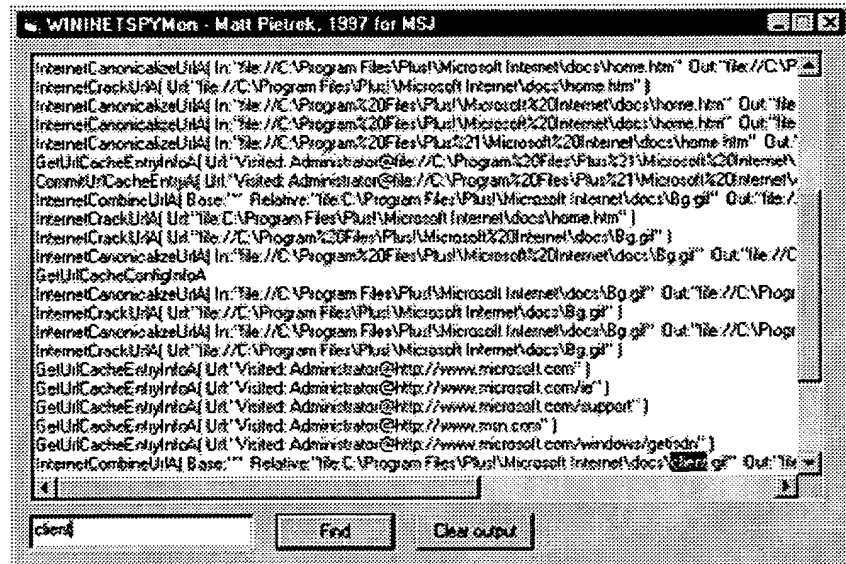


Figure 2 WININETSPYMon

While I won't go into all the details of WININETSPYMon here, two important procedures merit further commentary. In the Form_Load procedure, the code first creates the mailslot, which it names wininetspymon_mailslot. In the Timer1_Timer procedure, the code calls the GetMailslotInfo and ReadFile APIs in a loop until there are no more remaining messages. Each message (that is, line of output) is appended to the end of the edit control. The Timer1_Timer procedure is called every 50 milliseconds via the standard Visual Basic timer control.

The features of WININETSPYMon are mostly self-evident. To clear the edit control, click the Clear output button. To search for a string in the output, type the search text into the bottom edit control, then click the

Find button. You can click Find again to continue the search. The output edit control has the read-only attribute, but you can select and copy text out of it, allowing you to save some or all of the output to a disk-based file. I could have spent more time adding a lot more features, but WININETSPYMon is good enough for its intended purpose. If I'm going to spend time writing Visual Basic code, I want it to focus on more interesting things, like my investment analysis program.

To wrap up, here's a short list of things to keep in mind when setting up and using WininetSpy.

- Remember to run the WININETSPYMon installation program. This will make sure that you have the required Visual Basic 5.0 runtime library DLL and RICHTX32.OCX installed on your system.
- Copy the WininetSpy version of WININET.DLL to the same directory as IE. On my system running Windows NT 4.0 this is C:\Program Files\Plus!\Microsoft Internet.
- Start up WININETSPYMon before starting up IE. This is necessary because WININETSPYMon creates the mailslot that my WININET.DLL looks for in its DllMain.
- If IE doesn't work correctly (for example, it's unable to connect to the Web or display pages), the problem may be newer versions of the system DLLs such as URLMON.DLL or MSHTML.DLL. These DLLs may be importing additional functions from WININET.DLL that the logging version doesn't provide. The solution would be to add stubs for the missing functions. Unfortunately, I'm not aware of any reliable and easy-to-use method of determining which API stubs the system DLLs are looking for, but not finding.
- Don't forget to delete or move the logging WININET.DLL from the IE directory when you're finished spying. There's no sense in slowing down your system or risking funky behavior when you don't need to. For example, I wasted over an hour trying to figure out why I couldn't access a page that needed 128-bit encryption. It turned out an encryption DLL (SCHANNEL.DLL) determined that my WININET.DLL was not the same as the Microsoft version. It therefore refused to do 128-bit

encryption because, at the time of this writing, 128-bit encryption is still legally classified as a munition by the U.S. government. Happy spying!

Have a question about programming in Windows? Send it to Matt at mpietrek@tiac.com

From the September 1997 issue of Microsoft Systems Journal. Get it at your local newsstand, or better yet, subscribe.

© 1997 Microsoft Corporation. All rights reserved.
Terms of Use.

Contact Us | E-mail this Page | MSDN Flash Newsletter | Legal

©2003 Microsoft Corporation. All rights reserved. Terms of Use | Privacy Statement